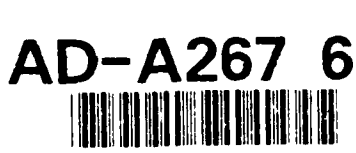
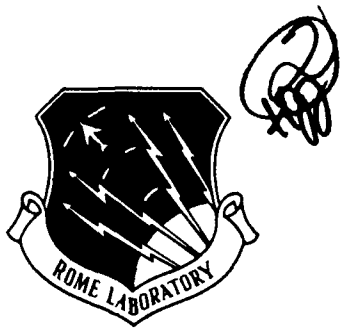
AD-A267 661



RL-TR-93-96
Final Technical Report
May 1993



NRC CLASS 1E DIGITAL COMPUTER SYSTEM GUIDELINES

SoHaR Incorporated

Herbert Hecht, Ann T. Tai, Kam S. Tso

DTIC
ELECTE
AUG 10 1993
S B D

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

63 8 9 040

93-18372



Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-93-96 has been reviewed and is approved for publication.

APPROVED: Joseph A. Caroli

JOSEPH A. CAROLI
Project Engineer

FOR THE COMMANDER: John J. Bart

JOHN J. BART
Chief, Reliability Sciences
Electromagnetics and Reliability Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (ERSR) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1993	Final Jul 91 - Jul 92

4. TITLE AND SUBTITLE
NRC CLASS 1E DIGITAL COMPUTER SYSTEM GUIDELINES

5. FUNDING NUMBERS
C - F30602-89-D-0100
Task 8
PE - 62702F
PR - R391
TA - QB
WU - 11

6. AUTHOR(S) Herbert Hecht, Ann T. Tai, Kam S. Tso

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SoHar Incorporated
8421 Wilshire Blvd, Suite 201
Beverly Hills CA 90211-3204

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory/ERSR
525 Brooks Road
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-93-96

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Joseph A. Caroli/ERSR/(315)330-4205.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report contains guidance on the design and development of reliable software for digital safety systems of nuclear power plants (Class 1E systems). The technology is dual-use in that both the nuclear and defense communities will benefit from it.

Four major topics are addressed: 1. Computer Programming Languages - C, C++, Ada and PL/M-86 were examined for applicability to high integrity safety systems; 2. Software Design and Development - Design and development process models, applicable standards, and Computer-Aided Software Engineering (CASE) tool usage for Class 1E systems development were identified; 3. Software Testing - This section addresses software test strategies and termination criteria for Class 1E systems; and 4. Fault Tolerance and Fault Avoidance techniques for developing highly reliable Class 1E system software are identified.

The Nuclear Regulatory Commission will use this development as a stepping stone leading to a draft nuclear regulatory guidance document. The DoD benefits by furthering research in software practices for high integrity systems.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software Quality, Software Reliability, Software Fault Intolerance, Nuclear Safety Systems, High Integrity Computing, Software Design, Software Testing	194
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	U/L

DISCLAIMER NOTICE

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, or any of their employees, makes any warranty, expressed or implied, or assumes any legal liability of responsibility for any third party's use, or the results of such use, of any information, apparatus, product or process disclosed in this report, or represents that its use by such third party would not infringe privately owned rights.

DTIC QUALITY INSPECTED 3

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

NOTES ON UTILIZATION OF THIS REPORT BY DOD AGENCIES

The report "Class 1E Digital System Studies" was prepared for use within the U. S. Nuclear Regulatory Commission, where the term "Class 1E" indicates an electronic system required for the safe shut-down of a nuclear plant under anomalous conditions or for the mitigation of the consequences of an accident. The report will be useful in its entirety to DoD agencies in the following areas:

1. Application-independent research in software practices for high integrity systems

2. Application-specific investigation where requirements are analogous to the nuclear plant shut-down situation, e. g., manufacture or handling of nuclear weapons or of other hazardous materials.

In addition, individual chapters of this report may be applicable as indicated below:

Chapter 2 - *A Comparative Study of Programming Languages*

While the Ada language will be selected for most DoD applications the comparisons presented in this chapter will still be of interest in (a) the revision of the Ada standard (Ada 9x) currently in process, and (b) the evolution of object oriented programming practices.

Chapter 3 - *Design and Development* Criteria

This chapter is broadly applicable to software development for high integrity systems. Section 3.2 - *Design and Development Process Models* -discusses problems of the waterfall model that are of particular interest to DoD agencies. In Section 3.3 - *Applicable Standards* - the description of the U. K. Ministry of Defense Standards 0055 and 0056, dealing with the application of digital systems to critical weapon systems, may be of special interest. The final two sections, tool usage and cost benefit analysis, should be useful to DoD agencies in their entirety.

Chapter 4 - *Test Methodology and Criteria*

This chapter is generally applicable in the DoD environment with the exception of Section 4.3 - *Management Considerations for Class 1E Software Test* - which is mostly oriented to specific concerns of the nuclear power field. In other sections some re-interpretation of terms may be necessary, particularly with regard to document nomenclature.

Chapter 5 - *Fault Tolerance and Fault Avoidance*

This chapter is generally applicable to the DoD environment. With regard to portions of Section 5.2 - *The Nature of Software Failures* - that deal with classification of failures DoD agencies will usually be governed by MIL-STD-1629 (Failure Modes, Effects and Criticality Analysis) or MIL-STD-882 (System Safety Programs). The military standards are presented here as one of several options, and the recommended classification scheme is in part based on them. Other sections are applicable in their entirety.

Contents

Chapter 1

Introduction 1
1.1 Overview of Report 1
1.2 Terms and Acronyms 2
1.2.1 Terms 2
1.2.2 Acronyms 11
1.3 Standards 12

Chapter 2

A Comparative Study of Programming Languages 15
2.1 Introduction 15
2.1.1 Overview 15
2.2 Stability of the Language Standard 16
2.2.1 C 16
2.2.2 C++ 16
2.2.3 Ada 16
2.2.4 PL/M-86 17
2.3 Program Control Directives 17
2.3.1 Conditional Control 17
2.3.2 Alternative Control 18
2.3.3 Iterative Control 18
2.3.4 Branching Control 19
2.4 Efficiency in Real-Time Operation 20
2.5 Strong Data Typing and the Ability to Check during Compilation and Execution . 20
2.6 Support for Object Oriented Programming 22
2.7 Cost/Benefit Data on the Use of Languages 24
2.8 Availability of CASE Tools 25
2.9 Summary and Conclusions 26

Chapter 3

Design and Development Criteria 28
3.1 Introduction 28

3.1.1 Overview of the Chapter 28
3.2 Design and Development Process Models 29
3.2.1 Waterfall and Spiral Models 29
3.2.2 Regulatory Concerns 31
3.2.3 Recommendations 33
3.3 Applicable Standards 33
3.3.1 Standards Utilization in the Survey 34
3.3.2 Current and Evolving U. S. Standards 34
3.3.2.1 U. S. Military Standards 34
3.3.2.2 Other U. S. Government Standards 36
3.3.2.3 U. S. Voluntary Standards 36
3.3.3 Current and Evolving International Standards 37
3.3.4 Foreign Standards 38
3.3.5 Regulatory Concerns 40
3.3.6 Recommendations 41
3.4 Tool Usage in Design and Development 42
3.5 Cost-Benefit Analyses 45
3.6 Conclusions 46

Chapter 4

Test Methodology and Criteria 48

4.1 Introduction 48
4.1.1 Overview of the Chapter 48
4.2 Technical Considerations for Class 1E Software Test 49
4.2.1 Testing in Software Lifecycle 49
4.2.1.1 Major Lifecycle Testing Activities 49
4.2.1.2 Objectives and Requirements 50
4.2.2 Software Test Strategies 55
4.2.2.1 Top Level Test Strategies 55
4.2.2.2 Lower Level Test Strategies 57
4.2.3 Test Termination Criteria 60
4.2.3.1 Test Termination Criteria Based on Coverage Measures 61
4.2.3.2 Test Termination Criteria Based on Statistical Measurements 62
4.3 Management Considerations for Class 1E Software Test 65
4.3.1 Qualifications of Test Personnel 65
4.3.1.1 Freedom from Bias 65
4.3.1.2 Procedural Qualifications 67
4.3.1.3 Familiarity with the Product and Application 68
4.3.2 Applicable Standards 69
4.3.2.1 The Role of Standards for Test Methodology 69
4.3.2.2 U. S. Military Standards 70

4.3.2.3 Other U. S. Government Standards 70
4.3.2.4 Non-Governmental Standards 70
4.3.2.5 International Standards 72
4.3.2.6 Foreign Standards 72
4.3.2.7 Evaluation of Test Standards 73
4.3.3 Software Test Tools 73
4.4 Criteria for Review of Class 1E Software Test Programs 75
4.4.1 Definition of the Application and its Environment 75
4.4.2 Testing in the System and Software Development Cycle 77
4.4.3 Management and Organization of Test Activities 77
4.4.4 Applicable Test and Documentation Standards 78
4.4.5 Test Methodology and Test Termination Criteria 79
4.4.6 Test Environment and Test Tools 80
4.4.7 Documentation and Review of Test Results 81
4.5 Conclusions and Recommendations 83

Chapter 5

Fault Tolerance and Fault Avoidance 85

5.1 Introduction 85
5.1.1 Motivation 85
5.1.2 Structure of this Chapter 85
5.2 The Nature of Software Failures 86
5.2.1 The Failure Process 86
5.2.2 Severity of Failures 88
5.2.3 Frequency and Criticality Classifications 90
5.2.4 Error Type Classification 91
5.3 Fault Tolerance 91
5.3.1 Architectures for Fault Tolerance 91
5.3.2 Error Detection 95
5.3.3 Integration of Fault Tolerance Provisions 98
5.3.4 Evaluation 98
5.4 Fault Avoidance 101
5.4.1 Enforcement of Good Software Engineering Practice 101
5.4.2 Formal Methods 103
5.4.2.1 Statement of the Problem 103
5.4.2.2 Current Capabilities 104
5.4.2.3 Recommended Regulatory Guidance 105
5.4.2.4 Discussion 106
5.5 Conclusions and Recommendations 111

Appendix A

CASE Tools Supporting Class 1E Software Process 113

A.1 Checklists of CASE Tools for the Design and Development Environment 113
A.1.1 CASE Tool Features 113
A.1.2 CASE Tools Checklist for the Design Input Documentation 117
A.1.3 CASE Tools Checklist for the Software Requirements Specification 119
A.1.4 CASE Tools Checklist for the Software Design Description 128
A.1.5 CASE Tools Checklist for the Code 134
A.2 Checklists of CASE Tools for Testing 142
A.2.1 Tools Supporting Complexity Measurement 142
A.2.2 Tools Supporting Syntax and Semantics Analysis 143
A.2.3 Tools Supporting Test Coverage Analysis 144
A.2.4 Tools Supporting Regression Test 144
A.2.5 Tools Supporting Test Data Generation 145

Appendix B

Examples of Fault-Tolerant Safety System Design 147

B.1 Background 147
B.2 Definitions and Assumptions 150
B.3 The Models 151
B.3.1 Notations 151
B.3.2 Failure Type-1 152
B.3.3 Failure Type-2 154
B.4 Comparisons of Schemes I and II 155
B.5 Scheme III - Extended Distributed Recovery Block (EDRB) 158
B.6 Techniques of Architecture Selection for Fault Tolerance 166

Appendix C

Survey Summary 169

Bibliography 172

List of Tables

2.1: Environment of Benchmarking Tests 20
2.3: Data Typing, Compilation and Run-Time Checking 22
2.5: Summary of Availability of CASE Tools 26
3.1: Tools Coverage 43
3.2: Cost Effects of Methodology Selection Criteria 45
3.3: Numerical Cost Effects 46
5.1: Criticality Ranking 90
5.3: Minimum Required Fault Tolerance Capability 100
A.1: Programming Language and Host System Information 117
A.2: Tool Features in Support of Design Input Documentation 119
A.3: Tool Features in Supporting Completeness of SRS 120
A.5: Tool Features in Supporting Consistency of SRS 122
A.7: Tool Features in Supporting Traceability of SRS 124
A.9: Tool Features in Supporting Robustness of SRS 126
A.11: Tool Features in Supporting Completeness of SDD 128
A.13: Tool Features in Supporting Robustness of SDD 130
A.14: Tool Features in Supporting Consistency of SDD 130
A.15: Tool Features in Supporting Verifiability of SDD 131
A.17: Tool Features in Supporting Traceability of SDD 133
A.18: Tool Features in Supporting Modularity of SDD 133
A.19: Tool Features in Supporting Understandability of SDD 134
A.21: Tool Features in Supporting Completeness of CODE 136
A.22: Tool Features in Supporting Correctness of CODE 136
A.23: Tool Features in Supporting Predictability and Robustness of CODE 137
A.24: Tool Features in Supporting Consistency of CODE 137
A.25: Tool Features in Supporting Structuredness of Code 138
A.26: Tool Features in Supporting Verifiability of CODE 138
A.28: Tool Features in Supporting Traceability of Code 140
A.29: Tool Features in Supporting Understandability of Code 140
A.31: Tools Supporting Complexity Measurement 143
A.32: Tools Supporting Syntax and Semantics Analysis 144
A.33: Tools Supporting Test Coverage Analysis 144
A.34: Tools Supporting Regression Testing 145
A.35: Tools Supporting Test Data Generation 146
A.36: Table A.36 Codes for CASE Tools.....146
B.1: Assignments of Parameters for Quantitative Evaluations.....152
C.1: Survey Summary --- Class 1E Products.....170

List of Figures

Figure 3.1: The Waterfall Model of the Software Life Cycle 30

Figure 3.2: Spiral Model of the Software Process 32

Figure 4.1: Reliability Estimation During Test 63

Figure 4.2: Input Profiles for Safety Systems 64

Figure 5.1: Failure Terminology 86

Figure 5.2: Failure Manifestations at Several Levels 87

Figure 5.3: Fault Tolerance Architecture 92

Figure 5.4: Examples of Formal Methods Languages 110

Figure B.1: Analog Implementation 148

Figure B.2: Scheme I System for Generation of Trip Signal 149

Figure B.3: Scheme II System for Generation of Trip Signal 150

Figure B.4: Software Structure for Scheme I 151

Figure B.5: Probability of Type-1 Failure as a Function of Hardware Failure 156

Figure B.6: Probability of Type-2 Failure as a Function of Software Failure 157

Figure B.7: Overview of the EDRB 158

Figure B.8: EDRB Operation (high level) 160

Figure B.9: EDRB Operation (low level) 161

Figure B.10: Fault Tree for Type-1 Failure (I) 162

Figure B.11: Fault Tree for Type-1 Failure (II) 163

Figure B.12: Comparison of Scheme I, Scheme II and Scheme III 164

Figure B.13: Type-1 Failure Probability Improvement of Scheme III over Scheme I 165

Figure B.14: Type-2 Failure Probability of Scheme I and Scheme III 167

Chapter 1

Introduction

1.1 Overview of Report

This document is furnished as part of the effort to develop NRC Class 1E Digital Computer System Guidelines which is Task 8 of USAF Rome Laboratories Contract F30602-89-D-0100. The report addresses four major topics, namely, computer programming languages, software design & development, software testing and fault tolerance and fault avoidance. The topics are intended as stepping stones leading to a Draft Regulatory Guide document. As part of this task a small scale survey of software fault avoidance and fault tolerance practices was conducted among vendors of nuclear safety related systems and among agencies that develop software for other applications demanding very high reliability (see Appendix C). The findings of the present report are in part based on the survey and in part on review of software literature relating to nuclear and other critical installations, as well as on the authors' experience in these areas.

The major headings of this report are:

- A comparative study of programming language (Chapter 2).
- Design and development criteria (Chapter 3).
- Test methodology and criteria (Chapter 4).
- Fault tolerance and fault avoidance (Chapter 5).

Because of the importance of terminology and standards in this report their listing follows this section, Section 1.3 lists standards referenced by this report. Appendices represent detail that supports the presentations of text sections, particularly with regard to tool usage and capabilities, and fault-tolerant system design.

1.2 Terms and Acronyms

1.2.1 Terms

Abstraction Denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

Acceptance test A series of system tests are performed on the delivered software and usually the acceptance of the software is made contingent upon the successful completion of these tests. This term is also used for fault-tolerant software or defensive programming in which acceptance test is the means of checking computational results for on-line error detection.

Alternative control Allows a program choose among taking multiple alternative actions according to the value of an expression; this structure contains an ordinary-valued expression followed by a list of the expression's potential values and the proper action to take for each one.

ANSI Standards Standards approved by the American National Standards Institute, Inc.

Availability Dependability with respect to the readiness for usage. Measure of correct service delivery with respect to the alteration of correct and incorrect service.

Backward recovery Form of error recovery where the erroneous state transformation consists of bringing the system back to a previously occupied state which is presumed to be correct.

Baseline A set of software items that has been formally reviewed and agreed upon; it then serves as the basis for further development, and can be changed only through change control procedures.

Benchmarking Objective comparison of languages, compilers, machines, and etc., by measurements against a program or standard.

Benign failure Failure whose penalties are small compared to the benefit provided by correct service delivery.

Black box testing An approach to devise test data without any knowledge of the software under test or any aspect of its structure.

Bottom-up testing In this approach, the lowest level modules in the system are tested first, by means of test-drivers. Next, modules are tested that connect to the lowest level modules, until the main module is included, culminating in the system test.

Branching control Unconditionally transfers control to another part of the program.

Branch testing A testing strategy which selects test data so that each predicate decision assumes a true and a false outcome at least once for a specified fraction of predicates during the test set execution.

CASE tools Computer aided tools for software engineering --- not an ad-hoc, grow-your-own programming method, but a well-defined, and well-recognized process.

Class A set of objects that share a common structure and a common behavior.

Code A uniquely identifiable sequence of instructions and data which is part of a module (e.g., main program, subroutine, and macro).

Common-mode failures Failures possibly caused by a single initiating event.

Comparison testing To detect test failures by comparing the output of two or more programs implemented to the same specification.

Compilation-time checking Checks performed during the compilation process.

Concurrency The property that distinguishes an active object from one that is not active. Concurrency is one of the fundamental elements of the object model.

Condition coverage The test is completed only when each condition in a compound predicate assumes all possible outcomes at least once.

Conditional control Allows a program choose between taking two alternative actions according to the result of a logical expression; this control structure can be nested.

Configuration item An aggregation of software or any of its discrete portions, that satisfies an end-use function and is designed for configuration management.

Configuration management The process of identifying configuration items, controlling changes, and maintaining the integrity and traceability of the configuration.

Correct service Service delivered in compliance with the system specification.

Corrective maintenance Preservation or improvement during its operational life of a system's ability to deliver a service complying with the specification. Fault removal during the operational life of a system.

Coverage Measure of the representativity of the situations to which a system is submitted during its validation compared to the actual situations it will be confronted with during its operational life. See also test coverage.

Critical Attribute of an entity whose improper behavior can cause or contribute to death and injury or major economic loss.

Criticality A measure derived from frequency and severity of failure modes.

Data flow testing The selection of test data that exercise certain paths from a point in a program where a variable is defined, to points at which that variable definition is subsequently used. By varying the required combinations of definitions and uses, a family of test data selection and adequacy criteria can be defined.

Decision coverage The test is completed only when every predicate decision in the program has executed a true and a false outcome at least once for a specified fraction of decisions.

Dependability Trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers.

Design diversity An approach to the production of systems, involving the provision of identical services from separate designs and implementations.

Design fault Fault of a software design that can be corrected only by a change of the design.

Domain testing A testing strategy involving the selection of test points that lie on and "just off" the boundaries of the subsets of the input domains (input data) associated with program paths.

Dormant fault Internal fault not activated by the computation process.

Dynamic analysis Analysis which requires execution of the program using input data.

Encapsulation The process of hiding all of the details of an object that do not contribute to characteristics essential in a given context.

Error compensation Form of error processing when erroneous state contains enough redundancy to enable correct service delivery.

Error detection The action of identifying that a system state is erroneous.

Error recovery Form of error processing where an error-free state is substituted for an erroneous state.

External fault Fault resulting from environmental interference or interaction.

Fail-safe system System whose failures can only be, or are to an acceptable extent, benign failures.

Fail-stop system System whose failures can only be, or are to an acceptable extent, stopping failures.

Failure The failure is an event in the computer when the content of a register transitions to an incorrect value (the value that results in the error).

Failure severity Grade of the failure consequences upon the environment.

Fault The cause of a software failure in the program.

Fault avoidance Methods and techniques aimed at producing a fault-free system. Fault prevention and fault removal.

Fault-based testing A testing strategy aimed at demonstrating that specific classes of faults are not in the program.

Fault diagnosis The action of determining the cause of an error in location and nature.

Fault passivation The actions taken in order to prevent a fault from being activated. E.g., to avoid to execute a program under heavy workload.

Fault removal Methods and techniques aimed at reducing the presence (number, seriousness) of faults.

Fault tolerance Methods and techniques aimed at providing a service complying with the specification in spite of faults.

Forward recovery Form of error recovery where the erroneous state transformation consists of finding a new state.

Formal methods The mathematically based techniques for describing system properties. The methods provide frameworks within which people can specify, develop and verify systems in a systematic, rather the *ad hoc*, manner.

Functional testing A black box approach in which the functional properties of the requirements or specifications are identified and test data selected to specifically test each of those functions.

Generic class A class that serves as a template for other classes, in which the template may be parameterized by other classes (objects, and/or operations). A generic class must be instantiated (its parameters filled in) before objects can be created. Generic classes are typically used as container classes.

Generic function An operation upon an object. A generic function of a class may be redefined in subclasses; thus for a given object, it is implemented through a set of methods declared in various classes related via their inheritance hierarchy.

Golden version Version of a program that is correct by definition.

Hard fault *or* **solid fault** Fault necessitating fault passivation.

Hazard analysis An iterative process composed of identification and evaluation of hazards associated with the computer system, to enable them to be eliminated or, if that is not practical, to assist in the reduction of the associated risks to an acceptable level.

Hierarchy A ranking or ordering of abstractions.

Inheritance A relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines a "kind of" hierarchy among classes in which a subclass inherits from one or more superclasses; a subclass typically augments or redefines the existing structure and behavior of its superclasses.

Impairments to dependability Undesired, but not unexpected, circumstances causing or resulting from un-dependability. Faults, errors, and failures.

Incorrect service Service delivered not in compliance with the system specification.

Independent faults Faults attributed to different causes.

Integration test A set of modules are tested together, ensuring that the combined specifications of these modules are met as the modules interact and communicate.

Integrity Condition of being unimpaired.

Intermittent fault Temporary internal fault. Faults whose conditions of activation cannot be reproduced or which occur rarely enough.

Internal fault Fault inside a system.

Intrusion Intentional operational external fault.

Iterative control Allows the program to loop over a certain set of instructions until a specified condition is met.

Latent error Error not recognized as such.

Maintainability Ease with which maintenance actions can be performed. Measure of continuous incorrect service delivery. Measure of the time to restoration from the last experienced failure.

Metaclass A class whose instances are themselves classes.

Missing path error A domain error where the conditional statement and computations associated with part of the input data domain are missing entirely, e.g., when data are absent or when negative values are input and only positive ones can be processed.

Modular redundancy Functionally equivalent computing elements execute the same task(s) in parallel in order to achieve error detection and/or masking.

Modularity The property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Multiple condition coverage The test criterion that all possible combinations of condition outcomes in each predicate are invoked at least once.

Mutation testing The construction of tests designed to distinguish between mutant (deliberately altered) programs that differ by a single mutation transformation (e.g. one symbol is changed).

Object An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable.

Object oriented programming A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

Operational fault Faults which appear during the system's operation.

Optimization A program compiled in a way that an attribute of the object (speed or size) is optimized.

Partition testing A testing strategy in which a program's input domain is divided into subdomains which are "the same" such that it is sufficient to randomly select an element from each subdomain in order to determine program faults.

Path testing A testing strategy which selects test data so that each path through a program or segment is traversed at least once during the test set execution.

Performability Performance-related measure of dependability.

Persistence The property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and/or space (i.e. the object's location moves from the address space in which it was created).

Polymorphism A concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass; thus any object denoted by this name is able to respond to some common set of operations in different ways.

Private A declaration that forms part of the interface of a class, object, or module; what is declared as private is not visible to any other classes, objects, or modules.

Predicate Every branch point of a program is associated with a predicate. This predicate evaluates to true of false, and this outcome determines which branch exit will be followed.

Preventive maintenance Corrective maintenance aimed at removing faults before they are activated.

Protected A declaration that forms part of the interface of a class, object, or module, but that is not visible to any other classes, objects, or modules except those that represent subclasses.

Public A declaration that forms part of the interface of a class, object, or module, and that is visible to all other classes, objects, and modules that have visibility to it.

Random testing A testing strategy which use realistic test data randomly chosen using distributions typical of what will be encountered in practice.

Random walk A random process that may be thought of as a particle moving among states in some state space. What is of interest is to identify the location of the particle in that state space. The salient feature of a random walk is that the next position the process occupies is equal to the previous position plus a random variable whose value is drawn independently from an arbitrary distribution.

Real-time function Function required to be fulfilled within finite time intervals dictated by the environment.

Real-time service Service required to be delivered within finite time intervals dictated by the environment.

Real-time system System fulfilling at least one real-time function or delivering at least one real-time service.

Regression testing Systematic repetition of testing to verify that only desired changes are present in the modified programs.

Related faults Faults attributed to a common cause.

Relational operator Operator which determine whether a certain relationship holds between the values of the left and right operands.

Reliability Dependability with respect to the continuity of service. Measure of continuous correct service delivery. Measure of the time to failure.

Robustness Quality attribute which refers to the extent to which a program continues to perform despite invalid inputs.

Run-time checking Checks performed during the execution of the program.

Safety 1) Dependability with respect to the non-occurrence of catastrophic failures. 2) Measures of continuous delivery of either correct service or incorrect service after benign failure. 3) Measure of the time to catastrophic failure.

Security Dependability with respect to the prevention of unauthorized access and/or handling of information.

Self-diagnostics The action of determining the cause of an error in location and nature taken by the object system itself.

Service System behavior as perceived by the system user.

Service restoration Transition from incorrect to correct service delivery.

Software System or application programs and their documentation.

Software diversity An approach to the production of software, involving the provision of identical services from separate software designs and implementation.

Software redundancy Extra programs provided to integrate the hardware redundancy into the complete fault tolerant system.

Spiral model A cyclical representation of software development process. Its inductive and synthetic nature accommodates reuse and prototyping. Its risk-driven approach facilitates the best mix of existing approaches to a given project.

Stability The property that a programming language is not subject to frequent changes.

Statement testing Every statement in the program is to be executed by the test data set at least once.

Static analysis It utilizes the computer program, examining this program for syntax errors and structural properties, but does not require execution of the program.

Strongly typed A characteristic of a programming language, according to which all expressions are guaranteed to be type-consistent.

Symbolic execution A testing approach where input variables assume symbolic values and by means of interpreting the program to express the output variables in terms of these symbols. These output variable expressions can then be examined to see if the program is computing the functions intended.

System test The entire software system is tested against the system specification.

Test coverage The fraction of elementary program structural elements (path, branch, etc.) that have been executed at least once by the test case set.

Test driver It is used to provide input data to the module under test that are normally furnished by other modules or external sources. The test driver together with the data collection means is sometimes referred to as a test harness.

Test oracle A means to automatically check the correctness of test output, e.g. a scientific simulation of the process.

Testability The property that a finite set of tests can be specified that will determine if the program that implements a function contains one of a specified set of faults.

Top-down testing In this approach, the main module is tested first. This is followed by integration tests involving modules called by or receiving data from this module, and this process continues until all modules are involved in a system test.

Transient fault Temporary physical external fault.

Trustability System's ability to provide users with information about service correctness.

Typing The enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways.

Unit test Each unit (or module, a component item in a program) is individually tested to ensure that its performance meets its stated specification. Also known as module test.

Unintended function The software performs some function which is not specified in the design.

User Another system (physical, human) interacting with the considered system.

Validation Methods and techniques intended to enable confidence to be reached in a system's ability to deliver a service complying with the specification.

Verification The process of determining whether a system adheres to properties (the verification conditions) which can be a) general, independent of the specification, or b) specific, deduced from the specification.

Waterfall model A model which stipulates that software be developed in successive stages. It emphasizes the feedback loops between stages and provides a guideline to confine the feedback loops to successive stages to minimize the expensive rework involved in feedback across many stages.

White box testing An approach which explicitly uses the program structure to develop test data.

1.2.2 Acronyms

ANS American Nuclear Society

ANSI American National Standards Institute

ASME American Society of Mechanical Engineers

DID Design Input Documentation

DOD-STD Department of Defense Standards

ESA European Space Agency

FAT Factory acceptance test

FIPS Federal Information Processing Standards

FRACAS Failure Reporting, Analysis and Corrective Action System

FRB Failure Review Board

IEC International Electro-technical Commission

IEEE The Institute of Electrical and Electronics Engineers, Inc.

ISA Instrument Society of America

METBF Mean execution time between failures

MTBF Mean time between failures

MIL-STD Military Standards

MOD Ministry of Defense (United Kingdom)

NIST National Institute of Standards and Technology

NRC Nuclear Regulatory Commission

NUREG/CR Nuclear Regulatory Commission Contractor Report

SAT Site acceptance test

SDD Software Design Description

SP Special Publication

SQA Software Quality Assurance

SRS Software Requirements Specification

SSCCSC System Safety-Critical Computer Software Components

V&V Verification and validation

1.3 Standards

The following is the list of standards referenced by a lower case single letter designator in the text of this report:

a. "Standard for Software Engineering of Safety Critical Software," Ontario Hydro/AECL CANDU, issued for one year trial use, 21 December 1990.

b. "Application Criteria for Programmable Digital Computer Systems in Safety Systems of Nuclear Power Generating Stations, "IEEE/ANS-7.4.3.2-1982, 1982.

c. "America National Standard for Information systems --- Programming Language C," ANSI X3.159, Dec. 1989.

d. "Quality Assurance Requirements for Nuclear Facility Applications," ASME-NQA-2a, the American Society of Mechanical Engineers, November 1990.

e. "Defense System Software Development," DOD-STD-2167A, 29 February 1988.

f. "Configuration Management Practices for Systems, Equipment, Munitions and Computer Software," MIL-STD-483A, 4 June 1985.

g. "Reliability Design Qualification and Production Acceptance Tests: Exponential Distribution," MIL-STD-781C, 1977.

h. "Technical Reviews and Audits for Systems, Equipments, and Computer Software," MIL-STD-1521B, Jun 1985.

i. "Procedures for Performing a Failure Mode, Effects and Criticality Analysis," MIL-STD-1629A, 24 November 1980.

j. "IEEE Standard for Software Test Documentation," IEEE Std 829-1983, 1983.

k. "Standard for Software Unit Testing," IEEE Std 1008-1985, Mar 1985.

l. "Standard for Software Verification and Validation Plans," IEEE 1012-1986, 1986.

m. "Software Safety Plans," IEEE P-1228, still under development (contact Cynthia Wright, MITRE Corp., McLean, VA. for more information).

n. "Reliability Program for Systems and Equipment Development and Production," (Notice 2), MIL-STD-785B, 5 August 1988.

o. "System Safety Program Requirements," MIL-STD-882B, 30 March 1984.

p. "Ada Programming Language," ANSI/MIL-STD-1815A, Jan. 1983.

q. "Guide to Software Acceptance," NIST SP 500-180, National Institute of Standards and Technology, April 1990.

r. "IEEE Standard for Software Quality Assurance Plans," STD 730-1981, 1981.

s. "Documentation of Computer Programs and Automated Data Systems," FIPS PUB 38, National Institute of Standards and Technology, 15 February 1976.

t. "Programmed Digital Computers Important to Sai ty for Nuclear Power Stations," IEC Std. 987, November 1989.

u. "Software for Computers in the Safety Systems of Nuclear Power Stations," IEC Publ. 880, 1986.

v. "Software for Computers in the Application of Industrial Safety-Related Systems," IEC 65A(Sec)94, 6 December 1989.

w. "The Procurement of Safety Critical Software in Defence Equipment," UK MOD 00-55, 5 April 1991.

x. "Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment," UK MOD 00-56, 5 April 1991.

Chapter 2

A Comparative Study of Programming Languages

2.1 Introduction

2.1.1 Overview

This chapter presents the results of a study of programming languages for digital Class 1E systems in nuclear power plants. It covers C, C++, Ada and PL/M-86, languages that were identified as being of interest for this application in a small scale survey conducted as part of this effort.

None of the languages investigated here have deficiencies which immediately preclude their use in digital Class 1E systems. The scarcity of CASE tools for PL/M makes that language less desirable than the others for new development. The lack of object-based and object-oriented controls for PL/M and C makes these languages less efficient in re-use and software maintenance. For newly developed software C++ and Ada offer more benefits than the other languages. The greater efficiency of C++ is likely to give it an advantage in future commercial software development.

The remainder of the chapter is organized as follows. Section 2.2 contrasts the stability of the language standards and Section 2.3 compares the adequacy and simplicity of program control directives. Section 2.4 investigates the real-time efficiency by presenting the benchmarking results. Section 2.5 contrasts the typing properties among the languages and Section 2.6 presents the features of the programming languages supporting object oriented programming. Section 2.7 discusses the cost/benefit data on the use of the languages. Section

2.9 summarizes the CASE tools available for different programming languages. The final section provides a brief summary of the findings.

2.2 Stability of the Language Standard

Stability means that a programming language is not subject to frequent changes. A programming language is said to have high stability if it has a firm standard.

2.2.1 C

C is a stable language.

- Classic (based on Kernighan and Ritchie), 1978, stable [1].
- ANSI Standard (ANSI X3.159-1989), stable [c].

2.2.2 C++

C++ is an evolving language [2].

- Release 1.0 (1985): the initial release which added basic object-oriented programming features to C, such as single inheritance and polymorphism, plus type checking and overloading.

- Release 2.0 (1989): improved upon the previous versions (1.0 and its minor releases) in variety ways, such as the introduction of multiple inheritance.

- Release 2.1 (1991): minor improvements upon 2.0.

Future versions are expected to support generic units and exception handling.

2.2.3 Ada

Ada has a firm and well policed standard, allowing neither supersets nor subsets.

- Ada83: the ANSI standard for Ada was released in 1983 (ANSI/MIL-STD-1815A, 1983) [p], stable.

- Ada9X: the project has been established to update the ANSI standards (draft being reviewed by public). Original language definition will probably change in a number of

small ways, involving clarifications, the filling of gaps, and the correction of errors [3]. A draft standard is scheduled to be produced by December 1992 and ANSI approval by September 1993 [4].

2.2.4 PL/M-86

The language PL/M-86 is based on PL/M Programmer's Guide, by Intel Corporation, 1987 [5]. It is an established language used on Intel microcomputers. PL/M programs are upward compatible across the 80[X]86 family of microprocessors.

2.3 Program Control Directives

The number of program control directives should be agreeable but restricted. All languages met this criterion. Program control directives can be classified into four categories:

- *conditional control* allows a program choose between taking two alternative actions according to the result of a logical expression; this control structure can be nested.

- *alternative control* allows a program choose among taking multiple alternative actions according to the value of an expression; this structure contains an ordinary-valued expression followed by a list of the expression's potential values and the proper action to take for each one.

- *iterative control* allows the program to loop over a certain set of instructions until a specified condition is met.

- *branching control* unconditionally transfers control to another part of the program.

All the programming languages under survey have adequate control directives to facilitate structured programming. In general, each control directive of one language can be mapped into an analogous one in other three languages.

The implementations of directives in different programming languages are illustrated below.

2.3.1 Conditional Control

Ada

```
IF condition THEN
```

```
        sequence_of_statements
    ELSEIF condition THEN
        sequence_of_statements
    ELSE
        sequence_of_statements
    END IF;

    EXIT loop_name WHEN condition        -- in loops
```

```
C, C++
    IF (condition)
        statement          -- statement can be a block
    ELSE
        statement
```

```
PL/M
    IF expression THEN
        statement;            -- statement can be a block
    ELSE
        statement;
```

2.3.2 Alternative Control

```
Ada -- expression must be discrete type

    CASE expression IS
        WHEN choice => sequence_of_statements
        WHEN OTHERS => sequence_of_statements
    END CASE;
```

```
C, C++ -- expression must be scalar type
    SWITCH (expression) {
        CASE constant-expression: statement
        DEFAULT: statement
    }
```

```
PL/M -- expression must be natural number
    DO CASE expression;
        case-0-statement;
        case-1-statement;
    END;
```

2.3.3 Iterative Control

```
Ada

    WHILE condition LOOP
        sequence_of_statements
```

```
        END LOOP;

        FOR i in 1..10 LOOP
            sequence_of_statements
        END LOOP;
```

C C++

```
        DO statement
        WHILE (expression)

        WHILE (expression)
            statement

        FOR (i = 1; i <=10; i++)
            statement
```

PL/M

```
        DO;
            sequence_of_statement;
        END;

        DO WHILE expression;
            sequence_of_statement;
        END;

        DO i = 1 TO 10 BY 1;
            sequence_of_statement;
        END;
```

2.3.4 Branching Control

Ada

```
        RETURN expression
        GOTO label
        RAISE exception
```

C, C++

```
        CONTINUE                                    -- in loop
        BREAK                                       -- in loop
        RETURN expression
        GOTO label
```

PL/M

```
RETURN expression
GOTO label
```

2.4 Efficiency in Real-Time Operation

Benchmarking tests were conducted in order to provide compilation time and execution time efficiency comparisons among the languages under survey. The environment of our benchmarks is shown in Table 2.1. The Sun SPARCstation 1 is a workstation rated at 12.5 MIPS running SunOS 4.1.

Table 2.1: Environment of Benchmarking Tests

Language	Machine	Compiler
C	Sun SPARCstation 1	/bin/cc (Classic C)
C++	Sun SPARCstation 1	Sun C++ Ver. 2.1 (AT&T Cfront Ver. 2.1)
Ada	Sun SPARCstation 1	SunAda 1.0 (Verdix 6.0)

The benchmark programs used are the Whetstone Benchmark [6]. The computation and execution times were measured with two optimization indexes, namely, minimal optimization and full optimization. Additional test cases are also used to compare the benchmark with full run-time check suppressed and no run-time check suppressed for Ada program. The results are summarized in Table 2.2. The unit used for measuring execution speed is KWIPS which stands for "Kilo Whetstone Iterations Per Second." Thus, a larger KWIPS represents a higher speed. The Whetstone program does not use C++ features and therefore the results of the C program are applicable to C++ as well.

The results show the greater efficiency of C (and C++) in both compile-time and run-time efficiency and the compactness of object code for these languages.

2.5 Strong Data Typing and the Ability to Check during Compilation and Execution

"Type" is defined as the domain of allowable values that an object may possess and the set of operations that may be performed upon the object. "Typing" refers to the enforcement of the class of an object, which prevents objects of different types from being interchanged or, at most, allows them to be interchanged only in very restricted ways. Typing is one of the fundamental

Table 2.2: Whetstone Ratings for Ada and C Languages

	Ada		C	
	No Optimization (No-Checks/Full-Check)	Full Optimization (No-Checks/Full-Check)	No Optimization	Full Optimization
Elapsed compile time (seconds)	37.1/40.1	123.8/127.5	2.7	9.9
Actual compile time (seconds)	29.2/30.2	108.1/117.3	2.2	8.2
Elapsed link time (seconds)	11.4/11.6	11.5/11.5	0.7	0.7
Actual link time (seconds)	6.0/6.3	6.1/6.0	0.4	0.5
Source code (bytes)	21423/21423	21423/21423	10329	10329
Object code (bytes)	311296/311296	311296/311296	24576	24576
Exec. speed (KWIPS)	3949/3422	6327/5962	3761	7916

elements of the object-oriented approach. "Strongly typed" is a characteristic of a programming language, according to which all expressions are guaranteed to be type-consistent.

C provides only limited compile-time type checking, and does not type check function arguments at all. On the other hand, the type checking features of C++ strongly encourage a programmer to declare the types of the arguments of all functions. Ada is a strongly typed language that performs all binding at compile-time. The types of all data are declared explicitly, and all decisions are made by the compiler at compile-time, not by objects at run-time. The information is summarized in Table 2.3.

Table 2.3: Data Typing, Compilation and Run-Time Checking

Languages	C	C++	Ada	PL/M
Strong data typing	No	Yes	Yes	No
Comp-time checking	Yes	Yes	Yes	Yes
Run-time checking	No	No	Yes	No

2.6 Support for Object Oriented Programming

Programming languages may be grouped into four generations, according to whether they support mathematic, algorithmic, data, or object-oriented abstractions. The most recent advances in programming languages have been due to the influence of the object model. A language is considered object-based if it directly supports data abstraction and classes; an object-oriented language is one that is object-based, but provides additional support for inheritance as a means of expressing hierarchies of classes. The language features for object-oriented programming are summarized in Table 2.4. The definitions of the terms in the table can be found in the glossary section of Chapter 1.

- C is neither an object-based nor object-oriented programming language.
- C++ is a "better C," in that it provides type checking, overloaded functions, and many other improvements. Most importantly, C++ is an object-oriented programming language.
- In the current definition of Ada, it is object-based, not object-oriented. However, a number of proposals would add object-oriented programming extensions to Ada (e.g., Classic Ada [7]).
- PL/M is neither an object-based nor object-oriented programming language.

Table 2.4: Language Features for Object-Oriented Programming

		C	C++	Ada	PL/M
Abstraction	Instance variables Instance methods Class variables Class methods	Yes Yes No No	Yes Yes Yes Yes	Yes Yes No No	Yes Yes No No
Encapsulation	Of variables Of methods	Public Public	Public, protected, private Public, protected, private	Public, private Public, private	Public Public
Modularity	Kinds of modules	Function	File (header/body)	Package (specification/body)	Procedure
Hierarchy	Inheritance Generic units Metaclasses	No No No	Multiple No No	No Yes No	No No No
Typing	Strongly typed Polymorphism	No No	Yes Yes (single)	Yes No	No No
Concurrency	Multitasking	Yes (defined by OS)	Yes (defined by class)	Yes (defined by language)	No
Persistence	Persistent objects	No	No	No	No

2.7 Cost/Benefit Data on the Use of Languages

Modern programming practices and available tool support are primary factors responsible for reducing programming cost and therefore making a language more effective. Tools are particularly valuable in the Class 1E environment because they collect information in a systematic way that facilitates review (this is of benefit to both the regulatory agency and the licensee). Even in the absence of specific supporting information a language that has good tool support can be presumed to possess a higher cost/benefit ratio than one that does not. This does not imply that the language for which the largest number of tools can be identified is necessarily the most cost-effective language because only a limited number of tools can be brought to bear on any given program. The key criteria for cost/benefit studies are therefore that (a) that an adequate number of tools be available, and (b) that the tools themselves be cost-effective. Tool availability is discussed in the Section 2.8. The cost effectiveness of tool usage is briefly addressed in the following.

The PA Computers and Telecommunications (PACTEL) organization conducted a tool benefits study for the Department of Trade and Industry in the United Kingdom [8]. The main purpose of the study was to quantify the UK market for software engineering tools and the benefits available from the use of formal development methodologies. Methodologies were grouped into two main classifications --- management techniques (project control, documentation/change control, and quality assurance) and system development techniques (system specification, software construction, testing, maintenance and amendment). The following conclusions were drawn from their results.

1. Automated methodologies show greater benefits than manually applied methodologies.

2. Management methodologies show a benefit only if automated.

3. Development methodologies generally show a benefit even when manually applied.

A survey was conducted for DoD by the Institute of Defense Analysis to investigate the areas of commercial Ada use in an effort to encourage better informed investments in the development and implementation of Ada products by the commercial community [9]. The findings of the study reveal that selection of Ada programming language is motivated by 1) increasing confidence in Ada technology, 2) better understanding of Ada, 3) rapidly increasing number of people who can successfully use Ada to solve a complex software problem, 4) availability of better Ada products exhibiting better performance and an improved ability to interface with other languages, and 5) DOD support for Ada. Positive aspects of Ada include: readability, reliability, and reduced test time. In addition, they found the following language features useful: package concept, exception handling, generics, flexibility of looping structure, strong typing, enumeration data types, and record data types. Negative aspects were: changing a specification part of a low-level package required time-consuming recompilation of a large

number of packages. Also, task switch time was too long, preventing the use of tasking in time-critical applications. The expansion ratio of Ada source code into memory was too large. Additionally, there was a lack of bit manipulation instructions, non-standard implementation of interrupts and non-standard implementation of embedding assembler language within an Ada system.

Recently, the Air Force released to the public a report of a business case study they conducted to determine under what circumstances a waiver to the DoD Ada requirement might be warranted for use of C++ [10]. The study concluded that both Ada and C++ represent improved vehicles for software engineering of higher quality products. The case study used in the report (the Command Center Processing and Display System --- Replacement) demonstrated development cost advantages for Ada on the order of 35% and maintenance cost advantages for Ada on the order of 70% under today's technologies. No reports from non-DoD sources on the advantages of Ada could be identified.

2.8 Availability of CASE Tools

Computer-aided software engineering tools substantially reduce or eliminate many of the design and development problems inherent in medium to large software projects. CASE tools employed in the early phases of software design and development yield lower costs and better results in the implementation and maintenance phases. This reduces the entire life-cycle cost.

CASE tools can be divided into the following five categories:

1. *Configuration management and version control* --- coordinate concurrent activities.
2. *Development environments* --- support requirement analysis, design, coding, debugging, etc.
3. *Static analyzers* --- assess the structural attributes of programs.
4. *Test tools* --- facilitate test data control, coverage measures, etc.
5. *Metrics tools* --- evaluate behavioral attributes of programs such as complexity.

Table 2.5 lists the number and type of CASE tools for each language that could be identified in this study. Further information on each tool is provided in the appendix. The applicability of tools to specific steps in software development will be discussed in the Design and Development Report.

Table 2.5: Summary of Availability of CASE Tools

Languages	Ada	C	C++	PL/M	Total
Configuration Control	2	3	1	0	6
Development Environment	16	16	10	2	44
Static Analyzer	8	5	1[1]	0	14
Metrics	9	4	0	1	14
Testing	18	12	7	1	38
Total	53	40	19	4	116

2.9 Summary and Conclusions

In a survey of vendors of nuclear plant protection systems and of government and industry organizations concerned with software for high integrity systems the following programming languages were identified as being of interest for current or future software efforts: Ada, C, C++, and PL/M. The study reported here compared attributes of these languages that are considered important for the Class 1E application.

The most significant findings of the survey are:

1. C, Ada, PL/M-86 are stable programming languages, while C++ is an evolving one.
2. All the four languages have adequate directives.
3. C and C++ are superior to Ada in compile-time and run-time efficiency.
4. C++ and Ada are strongly typed, while C and PL/M-86 are not.

[1]static analysis is also performed by the test tools.

5. C++ is an object-oriented language supporting object-oriented design directly, while Ada is an object-based language; C and PL/M-86 have no support for object oriented programming.

6. Ada and C have large numbers of available CASE tools, C++ has an adequate number of CASE tools; but PL/M-86 has very few.

Support for strong typing and object-oriented programming are particularly important for safety critical applications, and C++ is the only language that fully provides these attributes. The lack of standardization for C++ is expected to be remedied soon (the predominant commercial compiler constitutes a de facto standard). Ada's strong typing and object-based capabilities are acceptable for Class 1E applications, and a revision of Ada that will provide object-oriented capabilities is under consideration but without a firm schedule. Thus, C++ and Ada offer more benefits than the other languages for future Class 1E software development. The greater efficiency of C++ is likely to give it an advantage in future commercial software development.

Chapter 3

Design and Development Criteria

3.1 Introduction

3.1.1 Overview of the Chapter

Design and development of safety critical software can be controlled by a number of criteria that include:

a. the development cycle, including the definition of activities and of milestones that must be achieved to mark the conclusion of each activity.

b. item the methodologies used to develop the product and to monitor the process.

c. the required characteristics of the product.

d. the unique identification of intermediate and end items, including tools and support software (configuration control and configuration items).

Factor c. includes the language aspects (covered in the earlier chapter) and test criteria and fault tolerance provisions (to be covered in the later chapters), and thus product characteristics do not receive much attention here. The development cycle (factor a.) is discussed in Section 3.2 of this chapter. The other factors, b. and d. are discussed under the headings of standards and tools because these are the most effective means by which regulatory oversight over the essential characteristics of safety critical software can be achieved.

The major headings of this chapter are thus: Design and Development Process Models (Section 3.2), Applicable Standards (Section 3.3), Tool Usage (Section 3.4), and Cost/Benefit Analysis (Section 3.5). Appendix A represents detail that supports the presentations of text sections with regard to tool usage and capabilities.

3.2 Design and Development Process Models

3.2.1 Waterfall and Spiral Models

The primary activities that comprise the software design and development process include

- Establishing requirements
- Translating these into preliminary and complete designs
- Implementing the design in code
- Testing the code
- Integrating the tested code with hardware and/or other software.

Associated with each primary activity may be requirements for analyses, reviews, documentation, configuration control, and verification and validation. These requirements are of particular interest to regulatory agencies and are discussed in that context below.

Classical project management is based on completing and reviewing each of these activities prior to proceeding to the next one. The strictly sequential phasing of development activities is referred to as the *waterfall model* [11] and is the basis of most military project and software development standards, including MIL-STD-1521 (Reviews and Audits) and DOD-STD-2167 (Software Development). It is also used in ASME Standard,NQA-2 Part 2.7 (Quality Assurance Requirements of Computer Software for Nuclear Facility Applications). Figure 3.1 depicts one version of the waterfall model for software.

In complex projects requirements usually are initially defined only for the most important characteristics and must be supplemented to reflect design trade-offs and evolving user needs during later design and development phases. The process of requirements evolution is difficult to incorporate in a waterfall model and this difficulty, together with problems in applying a single activities progression to software that contains components of varying maturity (re-used or

Figure 3.1: The Waterfall Model of the Software Life Cycle

commercial software, modified code and completely new functions), has given rise to the *spiral model* of software development [12]. The spiral model (illustrated in Figure 3.2) assumes that requirements will be augmented several times during the course of a project and that other development activities will have to accommodate these changes. A typical spiral model provides for the following steps:

1. Skeletal requirements will be translated into a skeletal design which is primarily useful for identifying the most critical modules (from functional and performance perspectives).

2. After the capabilities of these modules are defined the requirements are enhanced and a more refined design is generated. This design may be implemented to the extent of allowing evaluation of functions that carry the highest risk.

3. Following evaluation of this initial design an additional set of requirements will be compiled that may permit design and implementation of a definitive version of the software.

While the spiral model maintains all of the key activities of the waterfall model it offers much more flexibility for their relative phasing, particularly for the purpose of allowing the requirements and the design to mature during a major portion of the development phase. A significant consequence is that it permits appropriate scheduling of documentation.

The spiral model is in wide use today. All Class 1E system vendors contacted in the survey were familiar with it, and most stated that it yields a more realistic representation of development than the waterfall model. The number of iterations (modifications of requirements or design) is not fixed, but at least the three stages described above are found in most current developments.

3.2.2 Regulatory Concerns

Both of the process models described above deal primarily with the information flow between a sponsor/user and the developer. The concerns of the NRC with respect to the development and design of Class 1E applications focus on the effect of the model on the visibility of quality, safety and stability aspects of the resulting software and system. The following are typical monitoring provisions that address regulatory concerns:

a. Analyses (failure modes and effects, fault tree, hazards).

b. Reviews (requirements, design, code, test readiness, test, and operational readiness).

Cut and Paste

Figure 3.2: Spiral Model of the Software Process

c. Documentation of the product (design, code, database, user and maintenance manuals) and of the process (development practices, design folders, test plans, procedures and results).

d. Configuration control (of requirements, design, code, and test cases).

e. Verification and validation (verification can be conducted at the end of each development phase, validation is usually conducted only as part of integration with hardware).

f. Quality assurance (exclusive of verification and validation) such as development and monitoring of project standards, and technical audits of software products and processes.

Because the waterfall model assumes a single pass through each of the primary development activities (requirements, design, etc.) there is a presumption that there is also a single instance of the associated monitoring steps (a-e above). When in actual system development iteration of a development activity is required, the need for repeating or updating a monitoring step is frequently overlooked or, although it is recognized, not accomplished because of lack of resources.

The spiral model inherently allows for iteration of primary development activities and thus facilitates more realistic scheduling of the monitoring steps. Obviously not every iteration will require implementation of all monitoring functions but the model provides a good framework for planning for preliminary and definitive versions of analyses and documents, and for specifying an appropriate scope of reviews, documents, configuration management and verification as development proceeds.

3.2.3 Recommendations

Because of its realistic representation of the development process and its compatibility for allowing refinement of documentation and other monitoring functions during the development it is recommended that the spiral model be used as the basis for regulatory review of software development for Class 1E systems.

3.3 Applicable Standards

To improve the flow of text standards are designated in the following by shortened titles and without reference to revision letters or dates. The standard listing in Chapter 1 contains the complete identification of the standards discussed here.

3.3.1 Standards Utilization in the Survey

All vendors contacted in the survey were familiar with and claimed ability to meet the requirements of IEEE/ANS Std. 7.4.3.2 "Application Criteria for Programmable Digital Computer Systems" and of ASME Std. NQA-2 Part 2.7 "Quality Assurance Requirements of Computer Software for Nuclear Facility Applications". The IEEE/ANS standard focuses on system rather than software development; the ASME standard deals with quality assurance. In spite of the recognized vagueness of the software development requirements incorporated in these documents there appears to be reluctance to adopt a more comprehensive document for industry-wide use. Most of the vendors were familiar with DOD-STD-2167A "Defense System Software Development" but saw no general applicability to Class 1E systems. However, all vendors stated that they had internal software development procedures that went considerably beyond the IEEE/ANS and ASME standards. None of these were made available for use in preparing regulatory guidance.

Related standards that were discussed by multiple vendors included:

MIL-STD-483	Configuration Management Practices [f]
MIL-STD-1521	Technical Reviews and Audits [h]
MIL-STD-1629	Failure Modes, Effects, and Criticality Analysis [i]
IEEE P-1228	Software Safety Plans (still under development). [m]

In most cases the discussion indicated readiness to apply only fairly narrow provisions of these standards. MIL-STD-1629 was discussed as being applied to hardware only and as needing considerable tailoring to include software.

A positive aspect of the currently utilized standards is that they cover both product and process aspects of the software, and that thus a precedent is in place for review of the software development process as part of regulatory approval. This facet of standards utilization is further discussed in Section 3.3.5 --- Regulatory Concerns.

3.3.2 Current and Evolving U. S. Standards

3.3.2.1 U. S. Military Standards

U. S. Military Standards pertinent to software development are listed below, together with a brief explanation of their relevance to the development of Class 1E software. A specific version of the standard is listed only where pertinent features have been added recently. An overall assessment of these standards is presented following the listing.

MIL-STD-785 Reliability Program for Systems and Equipment [n]

This is a hardware reliability standard which contains two key tasks that are valuable for evaluation of Class 1E software: Task 104 --- Failure reporting, analysis, and corrective action system (FRACAS), and Task 105 --- Failure Review Board (FRB). Combined hardware/software implementation of these tasks is desirable in that it prevents losing track of reports on failures for which it is not immediately obvious whether they are caused by hardware or software.

MIL-STD-882B System Safety Program Requirements [o]
+Notice 1

Notice 1 has added important software considerations to this originally exclusively hardware oriented document. Tasks 302 -- 307 include references to System Safety-Critical Computer Software Components (SSCCSC). Key provisions are identification of software elements in the top-level design hazards analysis, a requirements that code developers be made aware of the safety related functions implemented in the code, and for code level software hazards analysis, software/user interface analysis, and software change hazard analysis.

MIL-STD-1629 Procedures for Performing a Failure Modes, Effects, and Criticality Analysis (FMECA) [i]

Although primarily hardware oriented, this standard permits a "Functional Approach" to FMECA (par. 3.2) which permits applying it to software or to combined hardware/software entities. In addition to the tabulation of failure modes and effects (which is valuable for review of monitoring, circumvention and defense-in-depth provisions) the standard contains requirements for identifying single point failure modes and ranking failure modes by criticality. The FMECA can be made more valuable by requiring FRACAS (see MIL-STD-785 above) to list the applicable failure mode from FMECA, thereby identifying missing or incorrect listings in the FMECA.

MIL-STD-2167 Defense System Software Development [e]

This standard is aimed at software for mission-critical computers in an environment where software is developed by a contractor for use by military agencies. The standard has been criticized for being excessively detailed, requiring too much documentation, and being inconsistent with the use of the most recent programming languages (including Ada). All of these objections arise from provisions in Part 5 --- Detailed Requirements. However, the General Requirements of Part 4 (comprising about 9 pages) are much more broadly applicable and can provided valuable guidance in areas where the IEEE/ANS 7.4.3.2 standard is very vague. An example is Section 4.2 --- Software Engineering --- which requires that a documented development method be used, that a defined development environment be utilized, that development files be maintained, and that memory and processing reserves be specified.

The standards (or identified portions) listed above provide desirable augmentation of the IEEE/ANS and ASME standards currently accepted by vendors. The military standards referenced here are used by major software and systems vendors (outside the nuclear power field) and most were known to the nuclear system vendors who were contacted as part of the survey. In some cases automated tools for complying and/or checking compliance with the military standards exist. Reference to these standards (with tailoring where required) is therefore believed to be more cost-effective than developing separate and specific requirements for Class 1E software.

3.3.2.2 Other U. S. Government Standards

FIPS PUB 38 Documentation of Computer Programs and Automated Data Systems (National Institute of Standards and Technology) [s]

This document provides a very flexible outline for documentation requirements. For use with Class 1E software the applicable documents need to be identified and safety related headings added to some.

3.3.2.3 U. S. Voluntary Standards

IEEE P1228 Software Safety Plans [m]

Although the formal scope of this evolving standard is a safety plan (a document) it actually addresses requirements for a safety program and thus provides comprehensive guidance applicable to Class 1E software. It does not reference IEEE/ANS 7.4.3.2 or ASME NQA-2 but does not conflict with these. It references IEEE software standards for configuration management, quality assurance, test, etc. Potential weaknesses are (a) parameters required to make software acceptable are left to be specified in the plan, and (b) there is insufficient reference to utilization of software engineering tools.

IEEE 1012-1986 Standard Software Verification and Validation Plans [l]

This standard was written to provide direction to organizations responsible for preparing or assessing a Software Verification and Validation Plan. This standard may be used by project management, software developers, quality assurance organizations, purchasers, end users, maintainers, and verification and validation organizations. If V & V is performed by an independent group, then the SVVP should specify the criteria for maintaining the independence of the V & V effort from the software development and maintenance efforts.

3.3.3 Current and Evolving International Standards

IEC Std. 987 Programmed Digital Computers Important to Safety for Nuclear Power Stations [t]

This is a standard for hardware components only which may be of interest for format and project structure terminology.

IEC Publ. 880 Software for Computers in Safety Systems of Nuclear Power Stations [u]

This is a publication rather than a standard, and it contains some tutorial material, many recommendations, and numerous sections that sound like specific requirements but are weakened in the immediate or later context by allowing alternatives. It is fairly strong in software product requirements, e.g., par. 4.8 which requires self-supervision of the software and also supervision of the associated hardware. It takes a simplistic view of the software lifecycle and of the design process (e.g., requiring top-down design but also encouraging re-use of

existing software). It contains no requirements for software tool usage and makes no reference to formal verification or formal specifications.

IEC 65A(Sec)94 Draft Standard: Software for Computers in the Application of Industrial Safety Related Systems (Equivalent to BSI 89/33006)

This draft standard addresses software that performs safety-critical functions and also software in protection systems. The requirements are stated in fairly broad terms. Annexes of a tutorial nature are provided. It emphasizes self-checking provisions in the code and is also in other respects consistent with IEC Publ. 880 (see above).

3.3.4 Foreign Standards

UK MOD 00-55 Procurement of Safety Critical Software in Defence Equipment (Interim Standard) [w]

Safety critical software is defined as that which addresses the highest level of safety integrity defined in MOD 00-56 (see below). The standard consists of two parts: 1) Requirements and 2) Guidance; the Requirements are extremely rigorous but are somewhat relaxed in the Guidance volume. A Software Requirements Specification (plain language text) must be transformed into a Software Specification (formal notation, with commentary to relate it to the Software Requirements Specification). The formal specification must be proven to show correctness and consistency, and animated (transformed into an executable form, e.g., simulation) to show conformance to the Requirements Specification. Each subsequent step of software development (preliminary design, detail design, and coding) must be formally verified against the preceding step. Translation of a high level language program to object code must be accomplished by a compiler than meets the requirements of the standard (formally specified and verified), or the object code must be proven to be correct against the HLL code. There are also requirements for qualification of the staff, but these reduce to "adequate for the job". The Guidance acknowledges that there are few tools for any one of the steps and no complete environment for automating all of the required steps. Therefore "Rigorous Arguments" are allowed to be substituted for formal proofs for some verification activities. The Guidance indicates that the procedures are typically intended for programs that have fewer than 5,000 lines of code (the standard does not clarify whether this is source or object code).

UK MOD 00-56 Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment (interim Standard) [x]

The standard establishes severity and probability classifications for anomalous action in safety-critical systems and derives risk classification based on severity and probability. The highest risk (A) is associated with catastrophic events that have more than remote probability, critical events that have more than occasional probability, etc. The requirements of MOD 00-55 (see above) are inferred to be intended only for risk category A. A relaxation of requirements is possible where a given safety function can be accomplished by two or more independent means. In this case only one of the functions needs to meet the highest integrity requirements.

Ontario Hydro/ AECL CANDU Standard for Software Engineering of Safety Critical Software (issued for one year trial use 21 December 1990) [w]

This is a comprehensive standard for software engineering throughout the development phase. It describes general quality objectives of safety, functionality, reliability, maintainability and reviewability, and then decomposes these into 11 quality attributes, such as completeness, robustness, and understandability. General requirements for each development phase are stated in terms of the attributes required for key documents (software engineering products) that mark the end of the phase. Although the standard is based on a waterfall-like process model, the concept of requiring quality attributes to be evidenced in documentation is also applicable to the spiral development model. A desirable result of this approach is that the regulatory concern with control of the software development process (which is difficult to enforce) has been translated into requirements for deliverable products. Appendices to the standard give quite specific guidelines on the manner in which the attributes shall be demonstrated. As an example, the results of the Software Requirements Review are to be documented in a Software Requirements Specification (SRS), and the completeness attribute of the SRS is defined by 14 criteria of which the first three are that the SRS shall:

a. contain or refer to all requirements from the [systems specification] which are relevant to the software. This includes all functional, performance, safety, reliability, and maintainability requirements.

b. specify all additional requirements and software implementation design constraints.

c. identify those variables in the physical environment (i.e., temperatures, pressures, display readings, etc.) that the software must monitor and/or control and represent them by

mathematical variables. Monitored variables are those variables the software has to measure, and controlled variables are those the software is intended to control.

The standard is not a panacea for satisfactory software development (it contains practically no requirements for the development process proper) but it establishes concrete objectives that developers and regulatory agencies can agree on. A significant benefit of this approach is that the general as well as the specific requirements provide a framework against which methodology and tool capabilities can be evaluated. For this reason the standard has been selected as the outline for tools discussion later in this chapter.

3.3.5 Regulatory Concerns

It is desirable to base regulatory guidelines on existing standards because (a) a standard provides evidence of consensus and acceptance at least within a segment of the affected population, and (b) familiarity of vendors and users with the standard minimizes the cost associated with establishing and complying with the regulations. The combination of (a) and (b) also advances the availability of fully compliant software and systems.

If a viable international standard existed at this time it would receive serious consideration as the basis for regulatory guidelines because (a) the importance of permitting U. S. vendors to enter the export market without extensive re-engineering of their products, and (b) the increasing mutual dependence and collaboration of national licensing bodies. However, the documents listed under 2.3 do not singly or in combination represent a comprehensive international standard applicable to Class 1E software. The British MOD standards are targeted at defense equipment and currently do not have international standing. They contain many advanced requirements and it may be prudent to allow time for evaluation of the effort necessary for compliance as well as for the effectiveness of the resulting software before using them as a template for Class 1E applications.

A concept from the U.K. MOD Standard 00-56 that is significant to regulatory activities is the allocation of integrity requirements where two or more independent systems contribute to a safety function. Workable definitions for independence are presented in the document, and these as well as the allocation algorithm deserve consideration in future design, evaluation and regulatory efforts for safety-related software and systems.

Regulatory acceptance of hardware components usually makes some demands on control of the process (particularly configuration control) but important design characteristics, such as the number of independent channels and absence of common failure modes can be determined by on-site inspection and reference to installation drawings. In software the relative emphasis is reversed: the most sensitive attributes are embodied in the specification and design and become

increasingly difficult to examine in later product stages; inspection of the executable code present at the installation does not by itself constitute a significant part of the acceptance process. Thus, standards that form the basis for regulatory action must encompass the entire development cycle and the recent tendency is to focus particular attention on the requirements phase because faults introduced there are very difficult to detect and usually extremely costly to correct at the later stages.

In this regard it is encouraging that the need for control of the development process is accepted in all the significant standards cited above. However, except in the U.K. MOD Std. 00-55 and in the Ontario Hydro Standard, comprehensive and verifiable criteria for the software requirements and software specification are notable by their absence. Even the 00-55 document with its heavy emphasis on formal methods in the verification of the specification addresses completeness of the software requirements or misinterpretation of the requirements in arriving at the formal specification only in terms of "animation" (not a particularly exhaustive approach).

3.3.6 Recommendations

Because it explicitly addresses Class 1E systems and software and because of its acceptance by vendors and licensees the IEEE/ANS Std. 7.4.3.2 is recommended as one of the baseline documents for regulatory guidance. The standard is currently undergoing revision, and the working group is evaluating adoption of some of the features of IEC Publ. 880 which is also a specific nuclear reactor protection system document. The participants should be motivated to incorporate features identified as desirable by the international community for reasons discussed at the beginning of Section 2.5. The working group may want to direct its attention to at least the following material from the IEC publication.

- requirements for self-monitoring of the code and monitoring of the hardware status. This is not novel but it is more clearly formulated in Publ. 880 than in other standards documents.

- adoption of the document content criteria for Class 1E systems and software requirements, based on the Ontario Hydro Standard.

The revised IEEE/ANS standard can serve as a framework that references other U. S. standards, e.g., DOD-STD-2167A for selected documentation, MIL-STD-1521 for format and content of reviews, and MIL-STD-785 for failure reporting and failure review board tasks.

The IEEE P-1228 Software Safety Plans effort may eventually make contributions in its stated subject area and also in the conduct of the overall software safety program. However,

because it is still evolving and is not directly targeted at nuclear plants no specific recommendations for its inclusion in regulatory guidance are made.

The adoption of selected features of the U.K. MOD Std. 00-55 and 00-56 may best be handled as regulatory guidance from the NRC. The U. K. documents are not specifically targeted at nuclear reactor systems and thus interpretation and tailoring will be required. Particularly significant provisions from 00-56 are the risk classification and allocation of integrity criteria for multiple independent actuation of safety systems. In the 00-55 standard the emphasis on segregating modules that serve highest integrity functions from others appears very beneficial. The section on staff qualifications, though rather weak at present, may be adopted as a minimum requirement until more specific requirements can be coordinated. The reliance on formal methods is discussed in Chapter 5 of this report.

3.4 Tool Usage in Design and Development

Software tools can speed up the development and design process, reduce the amount of labor required, and make the quality less dependent on skills of the assigned staff. This latter benefit is of interest to regulatory agencies because personnel qualification criteria are difficult to implement and enforce. The vendors contacted in the survey were uniformly opposed to mandatory certification requirements for either the working level professionals or their supervisors. An additional advantage of tool based software development from the point of view of a regulatory body is that the documentation will be more uniform than that manually generated and thus more easily reviewed.

Against these arguments for tool usage must be weighed the considerable cost of tool usage, particularly of the introduction of tools. As with any piece of software, the more capable the tool the more difficult it is to learn to use it effectively. In addition, many tools require changes in the approach to design and development and affect many layers of the organization, including some that do not benefit directly from tool usage. For this reason there is frequently resistance to the use of tools and tool abandonment when initial expectations are not met, either due to limitations of the tool or due to inadequate training.

For Class 1E software the benefits from use of tools during design and development are expected to outweigh the disadvantages, particularly in the long run. The emphasis in the following paragraphs is therefore on the evaluation of the capabilities of tools against quality requirements arising during design and development of Class 1E software. As the frame of reference for these requirements the quality attributes listed in Appendix A.1.1 of the Ontario Hydro/CANDU Standard for Software Engineering of Safety Critical Software [a] has been used. The software quality attributes are identified for four documents that are associated with milestones in the software development process:

- Design Input Documentation (DID)
- Software Requirements Specification (SRS)
- Software Design Description (SDD)
- CODE.

The capabilities of a selected list of available software tools is evaluated against the requirements established in the Ontario Hydro standard for these documents. A classification of pertinent tool features and a description of the tools selected for this evaluation are presented in Appendix A.1.1. The analytical evaluation of tools
against the requirements for each of the four documents is shown in Appendices A.1.2-A.1.5.

The central issue under this topic is to determine the extent to which currently available tools can automate the generation of key documents required for review of the development process and of the delivered code. As shown in the following table, a very encouraging picture emerges.

Table 3.1: Tools Coverage

Deliverable	Total Number of Major Attributes	Number of Attributes Not Covered by Tools
Design Input Documentation (DID)	14	2
Software Requirement Spec (SRS)	8	1
Software Design Document (SDD)	9	0
Code	9	0
Total	40	3

In this tabulation an attribute is considered covered if major feature requirements arising from the attribute are covered. As an example, for the Consistency attribute in the SRS the following major feature requirements were considered (See Appendix A.1.3, Table A.5):

- Petri nets.
- State charts.

- Consistency checking.
- Scenario generation.

In this instance all features are supported by tools but it is still obvious that there remains a need for much analyst effort. The tools are particularly effective in taking on the repetitive parts of the job and in providing a framework for uniform, consistent input and output. These are very substantive benefits for both the developer and the regulators.

The attributes not covered by software tools are:

- Factor in relevant experience (for DID).
- Provide a clear definition of terms (for DID).
- Verifiability (for SRS).

The first two of these can be partially automated with ordinary database and/or word processing programs. The concept of verifiability involves traceability between levels which can be partially automated if a formal specification language and a coordinated tool is used.

A further question that arises under this heading is how many tools will be required to achieve the degree of automation that is indicated in Table 3.1. Here, again, very encouraging circumstances were encountered. A few of the tools are so capable that they can individually automate about 85% of the total attributes and up to 100% of the attributes for a given document. A combination of two or three tools can provide close to 100% attribute coverage (for the attributes indicated as covered in the above discussion). These findings are substantiated by the listing of specific tool capabilities in Appendices A.1.2 -- A.1.5.

Each appendix first identifies the attributes for the given document obtained from the Ontario Hydro standard. It then tabulates tools capabilities for each attribute and finally summarizes tools capabilities for all attributes identified for this document by a numerical score. This score is based solely on the number of tool features that support the automation of the given document. The information on tool feature was obtained from commercial brochures and bulletin boards. Within the scope of this effort the actual existence of the features could not be verified, nor was it intended to evaluate ease of use of tools or the quality of their output. The purpose of this study is to identify tool coverage, not to provide guidance for tool selection. Therefore neither the exclusion of a tool from the list of those evaluated nor a low numerical score in the summary tables should be taken as an indication of lack of suitability for Class 1E applications.

3.5 Cost-Benefit Analyses

The topics discussed in this chapter deal with the design and development methodology for Class 1E software and the conclusions drawn from this discussion may affect the development cost. An analysis of the potential cost-benefit relations is therefore presented below.

As a basis for this assessment it is assumed that equipment containing Class 1E software will not be licensed unless the NRC is certain that it meets its safety criteria. Therefore the trade-offs are not between development methodologies that yield different levels of safety or reliability but rather between the cost incurred by various methods in demonstrating that the licensing criteria have been met. Where failure consequences are affected, their economic impact can be evaluated from data presented in [58].

The costs to be considered are those borne by the licensee and those incurred by the NRC. The software developer's costs are assumed to be passed on to the licensee and are therefore treated as part of the licensee's cost. The most direct way in which NRC actions will affect the total cost is by issuance of a Regulatory Guide that endorses a given standard and/or requires submission of specified documents. The underlying cost driver is the latitude permitted in selecting design and development methodologies as indicated in the following table.

Table 3.2: Cost Effects of Methodology Selection Criteria

Cost Element	Loose Criteria	Intermediate Criteria	Strict Criteria
Licensee: Development Cost	Low	Medium	High
Licensee: Delay of Approval	High	Medium	Low
NRC: Cost of Processing	High	Medium	Low

An example of loose criteria is the present version of IEEE/ANS 7.4.3.2 without additional guidance. An example of intermediate criteria is the Ontario Hydro standard because it has tight requirements on deliverables but very few on the means for generating them. An example of strict criteria is the UK MOD Std0055 which requires specific methodologies.

The increase in development cost as criteria are tightened is due to the developer having to adapt to new practices, possible denial of commonality with other applications of the same software, and, particularly in the case of strict criteria, forcing the use of an inherently higher cost approach. The delay of approval cost is directly associated with the processing cost within the NRC. With loose criteria a given application requires much more staff effort and time for review, and during that interval the licensee must wait for approval. Also factored in the delay

of approval cost element is the expected cost of modifications to the software or the installation that may be required to obtain a license. Loose criteria introduce uncertainty into the licensing process that is undesirable for all parties.

The following numerical examples consider two software development scenarios: comparatively low unit development cost for software that has many Class 1E applications, and comparatively high development cost for software that has only a few Class 1E applications. The delay of approval cost and the NRC processing costs (both as identified in the proceeding paragraph) are held constant. The examples therefore serve as a general sensitivity study with respect to the ratio of increasing and decreasing costs. In the following table the low and high unit development cost assumptions are separated by a slash (/).

Table 3.3: Numerical Cost Effects

Cost Element	Loose Criteria	Intermediate Criteria	Strict Criteria
Licensee: Development Cost	2/20	4/40	8/80
Licensee: Delay of Approval	20	10	5
NRC: Cost of Processing	10	5	3
Total	32/50	19/55	16/88

It is seen that loose criteria do minimize total costs for the high unit cost assumption and strict criteria minimize the total cost in the case of low unit cost. However, the cost differential between intermediate criteria and those having the lowest cost in each case is very small and may vanish if a realistic assessment about the cost apportionment is introduced (for loose criteria and high unit cost the delay may be more costly than indicated and for strict development criteria the cost of customizing standard software will probably be higher than allowed for). It is therefore concluded that intermediate criteria represent a cost-effective basis for regulatory requirements.

3.6 Conclusions

Three major factors that affect the design and development of Class 1E software were investigated in this chapter:

- the life cycle model.
- current applicable standards.

- design and development tools.

In addition, cost-benefit relations for a regulatory approach were evaluated.

In the study of life cycle models it was found that the spiral model of software development was the most suitable one because it recognizes that requirements are evolving during development rather than being fixed at the outset. This acknowledgement of evolving requirements permits realistic scheduling of milestones for delivery of documents for review of the development process.

In the standards area it is concluded that IEEE/ANS Standard 7.4.3.2-82 [b] is the most widely accepted one and has the potential of serving as a starting point for regulatory guidance. However, its present content provides wide latitude for the development process as well as for the delivered product. The current standard therefore does not provide assurance that compliance with it will assure suitability of software for Class 1E applications. A recent standard created under the auspices of Ontario Hydro [a] incorporates review of the development process into four deliverable products each one of which must meet reasonably well defined criteria. This standard is still being evaluated in Canada but appears to be a good statement of requirements that regulatory agencies can levy on software development and that developers can meet with available technology. As explained in connection with Table 3.3 this standard has achieved an efficient balance between loose and excessively tight controls on software development.

The evaluation of design and development tools showed more than a half dozen products that can make substantive contributions to the development process. Several of the tools studied could individually compile information that satisfies the majority of the requirements established in the Ontario Hydro standard, and a combination of two or at most three of these tools could address all requirements for which a software development tool could reasonably be applied. Therefore regulatory guidance should be formulated to be readily compatible with tool use although it may not be desirable to make tool use mandatory (and certainly not the use of specific tools).

The cost-benefit assessment showed that an intermediate level of control over the development process will probably lead to minimum cost for the community that comprises both licensees and the NRC. Loose criteria minimize the development cost and this leads to low overall cost when only very few copies of the software are installed, and very strict criteria may lower cost when many copies of exactly the same program are installed. But both of the extreme approaches have very adverse cost consequences if the assumptions about the number of users are not met. The intermediate approach is relatively insensitive to the number of installations and is recommended. The Ontario Hydro standard is regarded as an example of intermediate criteria.

Chapter 4

Test Methodology and Criteria

4.1 Introduction

4.1.1 Overview of the Chapter

Software testing is a dynamic verification technique that exercises the software by supplying it with input values usually selected by the verifier. The term dynamic refers to changes (due to selected input values) in the data controlled by the software under test; it distinguishes testing from other verification techniques, such as design or code audits, in which agreement with the stated requirements of a preceding phase is assessed by informal or formal reasoning. Testing is frequently considered to be a part of the quality assurance activities. This is usually true for the late stages of test but not necessarily of the early ones, such as unit test. Also, within this document the generation of test plans and specifications is considered within the scope of test, and this is not usually the responsibility of the quality assurance function.

Testing is always partial as it is not feasible to exercise software with each possible sample from the whole input domain. The tester is then faced with the problem of selecting a subset of the input domain that can best reveal the expected but unknown faults, and determining when the test can be terminated. Test of safety critical software will be guided by a number of subjective considerations that include:

a. Test activities in the software lifecycle.

b. Selection of strategies used to test the product.

c. Test termination criteria.

These three topics form the major subdivisions of the section on technical considerations. The major headings for the body of this chapter are:

Section 4.2: Technical considerations for Class 1E software test (this addresses the three topics mentioned above)

Section 4.3: Management considerations for Class 1E software test (including qualifications of test personnel, selection of standards, and tools and support software)

Section 4.4: Criteria for review of Class 1E software test programs

Section 4.5: Conclusions and recommendations

Sections 4.2 and 4.3 provide general background while Section 4.4 addresses specific needs of reviewers of test programs.

4.2 Technical Considerations for Class 1E Software Test

4.2.1 Testing in Software Lifecycle

The software lifecycle involves a sequence of managerial and technical activities which can be grouped in phases as discussed in Section 3.2. Although the phases tend to overlap and the process can be iterative, there is always a moment in which a specific phase is formally concluded with the acceptance of the deliverable items.

Software test affects activities in all of the software development phases. As a lifecycle process, software test supports:

- Early detection of software problems.
- Preparation of appropriate test facilities.
- Consideration of the user interface requirements during software development.

4.2.1.1 Major Lifecycle Testing Activities

The major activities for testing in the software lifecycle are shown in Table 4.1.

It is seen from the table that tests are executed in the inverse order in which the planning takes place. Particularly long time intervals will usually elapse between the planning for the system (combined hardware and software) test and its execution, and between the planning for the software acceptance test and its execution. Project management must allow for updating of the requirements and plans for these tests at regular intervals in order to have current documents available for the test teams. To minimize the need for updates the preparation of the test procedures (detailed documents that govern equipment configuration, test case sequence, and interpretation of test outcomes) can be deferred until shortly before the test is to be conducted. The sequence of tests shown in Table 4.1 is consistent with that prescribed in DOD-STD-2167A [e] with the following nomenclature changes: software acceptance test instead of CSCI test in the DOD standard, and system test instead of system integration and testing in the DOD standard. The system test is usually an essential part of the validation procedure and is in some documents referred to as validation test.

The OH/AECL Standard for Software Engineering of Safety Critical Software [a] adds a requirement for a reliability test which is intended to be conducted after the system (validation) test. The general planning for the reliability test can be accomplished concurrent with that for the software acceptance test, and this should be augmented by detailed planning after the software structure has been completely defined, typically after coding. The methodology employed for the reliability test is largely identical with that of the system test (test scenarios selected from expected usage environment) but test duration and random parameter selection are oriented toward obtaining statistically significant test results.

Where the spiral model for software development is used, the applicable test sequence is repeated for every pass through the spiral. During early cycles through the spiral the systems, acceptance and integration test requirements may be abbreviated or even absent but during the final pass all will need to be documented and executed.

Regression testing is not covered separately in this presentation because the methodology and criteria used are based on those of the test that is being repeated. Thus, regression testing at the unit level will use criteria described here for unit testing, and regression testing at the software integration level will use the criteria described here for software integration testing.

4.2.1.2 Objectives and Requirements

The objectives and requirements for each test are as follows:

Unit Testing The objectives of unit testing are to verify that implementation of software units, modules, and subelements, is consistent with the Software Design Document (SDD), that the

Table 4.1: Lifecycle Testing Activities

Lifecycle Phase	Testing Activities	Products
Requirements	Begin planning for system testing Begin planning for software acceptance testing	System Test Plan Acceptance Test Plan
Design	Begin planning for software integration testing Begin planning for software unit testing	Unit Test Plan Integration Test Plan
Implementation	Software unit test execution	Unit Test Report
Test	Software integration test execution Software acceptance test execution System test execution	Integration Test Report System Test Report Acceptance Test Report
Installation	Final test report preparation	Final Test Report

units do not perform unintended functions, and that their interfaces behave as specified in the SDD.

All of the following requirements for unit testing of Class 1E software shall be met by modules directly involved in a protective function. The overall purpose of these requirements is to exercise the program for all conditions that may cause incorrect or unintended system operation. Abbreviated requirements may be used for support modules (report generation, logging, etc.).

- Each unit shall have a Software Unit Test Specification which the module shall be tested against.

- Functional unit test shall define a sufficient number of structural test cases, derived from the analysis of the SDD, to ensure that the executable code for each unit behaves as specified in the SDD. The number of test cases shall be considered sufficient when they include:

 - all possible outcomes for decisions derived from the SDD,

 - all possible conditions for each decision,

 - test on each boundary and values on each side of each boundary for each input.

 If the functional unit test is conducted with a dynamic analysis tool, the results of the functional test can be used to satisfy many requirements of the structural and special values test defined below.

- Unit test shall define a sufficient number of test cases, derived from the analysis of the code, to ensure that the executable code for each unit behaves as specified in the SDD. The number of test cases shall be considered sufficient when they cause to be executed at least once:

 - all possible conditions for each decision (this also assures executions of each statement and of each decision outcome),

 - each loop with minimum, maximum, and at least one intermediate number of repetitions,

 - all possible paths with loops with zero and one execution,

 - cause a read and write to every memory location used for variable data

 - cause a read of every memory location used for constant data

- Unit test shall include test cases to cause each interface requirement to be exercised.

These requirements are substantially consistent with those in Appendix C of [a]. Higher assurance of correct behavior and the absence of unintended functions can be achieved by path testing (see Section 4.2.2.2). This requires many more test cases and should be reserved for functions requiring highest integrity.

Software Integration Testing The objective of software integration testing is to verify that all functional and performance requirements for the integrated software are met. Testing of interfaces and of data flow between modules is essential to meeting this objective but very little specific guidance for these steps could be found in the literature. The following requirements are therefore largely based on the author's experience.

- Interfaces shall be exercised with a sufficient number of combinations of parameters to insure functioning under all operational and exception states. The following interfaces are included in this requirement:

 between developed software modules or units
 between developed software units and non-developed (off-the-shelf) components
 between non-developed software components with external software or devices (simulated as necessary)

- Calling sequences shall include cases of missing and illegal parameters.

- Called units or devices shall be exercised with simulated busy and inoperative states and with check code error conditions where these may be encountered.

- All modes of data transfer between modules shall be exercised

For very large software products integration testing may be conducted in phases, progressing from small aggregates of modules to larger ones. In some documents the term subsystem test or subsystem integration test is used for the initial phases of the software integration test.

For highly time critical systems, performance issues may also be included in the integration test, and for systems where security is very important this issue may also be addressed. For the broader class of systems these areas are assumed to be tested as part of the software acceptance test.

Acceptance Testing The objective of the software acceptance test is to determine that the software product complies with the specification. Since the specification defines not only the required software functions but also performance (speed of execution) and other attributes the acceptance test must be designed to include the entire scope of the specification. According to a recently published Guide to Software Acceptance [13] the following categories are typically covered:

- Functionality
- Performance
- Interface Quality
- Overall Software Quality
- Security
- Software Safety

Consistent with the above stated objective, the detailed requirements for the test are derived from the software specification. For some of the categories audits or reviews may be used to supplement or replace test. If neither the code nor the test environment has significantly changed between the integration test and the acceptance test the results of the former can be used to partially satisfy the latter.

System Testing The objective of system testing is to validate the entire program against system requirements and performance objectives. It is in some documents referred to as validation testing. It is conducted on target hardware interfacing with an actual or simulated plant environment.

The following requirements have been adopted from [a].

- define test cases to test each functional requirement in the System Requirements Specifications (SRS).
- define test cases to test the performance requirements as described in the SRS.
- define test cases to exercise any interfaces between the software and the target hardware, and its environment.

- define test cases to test the ability of the system to respond as indicated in the SRS to software, hardware, and external errors.

- define test cases, using dynamic simulation of input signals, to cover normal operation, anticipated operational occurrences, abnormal incidents and accident conditions.

4.2.2 Software Test Strategies

4.2.2.1 Top Level Test Strategies

The following paragraphs present an overview of software test strategies that have been discussed in the recent literature. In the opinion of the authors of this report some of these strategies have little or no immediate applicability to the testing of Class 1E software but they are included to permit each reader to formulate an independent assessment. Recommendations for the evaluation of Class 1E test programs are presented later.

Functional Testing Functional testing [14] is popular in industrial and commercial software applications. It is a black box approach in which the functional properties of the requirements or specifications are identified and test data selected to specifically test each of those functions.

There are two problems with this approach. First, although requirements and specifications provide many meaningful functions that can be the focus for functional testing, software may contain a much richer collection of functions than those put forth in specifications. Moreover, specifications are often inadequate for providing the detailed information required for testing. Another problem has been the lack of any unifying and fundamental theoretical basis for such testing. Thus the industrial efforts using functional testing are both ad hoc and incomplete in scope. Nevertheless, functional testing is appropriate as a first step in the testing of Class 1E software.

Structural Testing Structural testing is a white box approach in which explicit knowledge of the software under test and its structure is used to generate test data and to evaluate the thoroughness of test [15]. Structures such as branch and path are determined by program control flow. The fraction of program constructs that is structurally exercised is called "coverage." Structural testing aims to detecting discrepancies between functional specification and software implementation by exercising a program based on its structural properties.

Research has shown that there are inherent limitations to the use of structural testing alone, for test data based only upon the software code and structure will fail to detect the absence

of certain features that might be missing in the software. Information and the associated test data must be derived from the software specifications, design documents, or from some other source.

Most current standards for critical software require a combination of functional and structural testing.

Random Testing Random testing is a black box approach in which a random value is selected for each input variable of the program and each test data point then consists of the collection of these. The randomness of the test data makes the technique potentially useful for discovering unintended functions.

The random data may come from a uniform or normal distribution covering the entire range, or from a distribution that randomizes around boundaries of routine and disturbed operation. The random walk process in which small increments from the initial condition are generated is also a useful means for generating test data. Data from known or suspected prior failures may be used to define the initial conditions.

In the Software Testing and Evaluation Methods (STEM) experiment at the Halden Reactor Project uniform random distributions over the entire input domain were found more efficient in achieving branch coverage than "systematic" methods (based on requirements, etc.) [16]. The uniform random data also were effective in finding faults. In the STEM experiment a 'golden version' of the code was available for generating correct output and this facilitated positive identification of failures. By this means 95% of all natural faults and 99% of seeded faults were detected.

With knowledge of the STEM findings, the editor of the Techniques Summary of the series "Dependability of Critical Computer Systems" [17] concludes:

> There is some evidence that this test data strategy [uniform random inputs] is quite effective, giving similar levels of branch coverage, statement coverage and fault detection efficiency to systematic methods. The main problem is to determine which tests result in a failure, so this technique is mostly used in conjunction with comparison testing.

The major advantages cited include:

- No program and specification analysis is required.
- A large number of test data sets can be produced with little effort.

- Not prone to human bias and 'mind-set'. For example, systematic tests may check that outputs are set, but not that they are cleared.

The major disadvantage cited is the difficulty of determining the correct response to each randomly generated data set. This difficulty can be overcome where a 'golden version' of the software exists that can be used for comparison or through use of a plant simulator which signals the existence of safe and unsafe states. See also "Evaluation of Random Testing" [18]. The applicability of random testing in the Class 1E environment is discussed later.

4.2.2.2 Lower Level Test Strategies

The following lower level test strategies identify the detailed procedures to be used in the implementation of the higher level strategies. Definition of a lower level strategy is particularly important for structural testing where the intensity of the test effort and effectiveness of coverage are determined by the lower level selection. The probability of uncovering unintended functions increases significantly in going from statement coverage to condition coverage and increases greatly by going from condition coverage to path coverage. For these reasons knowledge about these strategies is important for the evaluation of Class 1E software test plans and environments.

Statement Coverage With statement coverage, every statement in the program is to be executed by the test set at least once. Unless one encounters reachability problems, this is certainly a requisite of a test plan. It is not nearly strong enough, however, for example consider the statement

```
IF X > 0 THEN S;
```

In this case, we assume a null ELSE statement; thus with statement coverage, the ELSE condition might never be checked and yet contribute to a serious error. Statement coverage by itself is not adequate for class 1E software testing.

Branch Testing In branch testing (or decision coverage) each predicate decision assumes a true and a false outcome at least once during the test set execution (or each possible outcome for a CASE statement). This coverage criterion clearly overcomes the problem with the null ELSE example previously given, but there are problems with decision coverage as well.

For example, consider a program with two successive IF-THEN-ELSE constructs; if tests are selected which execute the THEN-THEN alternative of these predicates, as well as the ELSE-ELSE alternative, then this criterion is satisfied. Yet the THEN-ELSE alternative is not

adequately tested, and might well be in error. Complete branch coverage also provides complete statement coverage.

Condition Coverage Another weakness in branch testing is encountered with compound predicates such as

$$\text{IF } (A > 0) \text{ AND } (B < 5)$$

Branch testing will treat this compound predicate the same as a simple predicate, testing only for true and false outcomes and ignoring the fact that a false outcome could occur from two distinct Boolean clauses. For this reason, a condition coverage will require that, during test set execution, each condition in a compound predicate assumes all possible outcomes at least once. Complete condition coverage also provides complete branch coverage.

Path Testing A path is any sequence of statements that can be traversed in the execution of a program or program segment. Path testing is based on the use of the control flow of the program. Path testing involves two operations:

1. selection of a path or set of paths along which testing is to be conducted,
2. selection of input data to serve as test cases, which will cause the chosen paths to be executed.

Because of the presence of iteration loops, there is potentially an infinite number of distinct paths in a program. Even in a program without iteration loops, the number of distinct paths, is an exponential function of the number of predicates in the program. Thus, for any nontrivial program it is very difficult to generate test data for all paths in that program.

One of the advantages of path testing is that it tends to be easier to automate than other approaches, such as functional testing. Complete path coverage also provides complete condition coverage.

Data Flow Testing Data flow testing selects test data that exercise certain paths from a point in a program where a variable is defined, to points at which that variable definition is subsequently used. By varying the required combinations of data definitions and uses, a family of test data selection and adequacy criteria was defined [19]. The data flow criteria can be used to bridge the gap between the requirement that every branch be traversed and the frequently

impossible requirement that every path be traversed. The criteria focus on the interaction of portions of the program linked by the flow of data rather than solely by the flow of control. Thus, not only do the criteria fall in between branch testing and path testing in terms of difficulty of fulfillment, they also guide us in the intelligent selection of paths for testing. Data flow testing is potentially useful as a component of Class 1E software testing but is not currently identified in any major standard and is not automated.

Partition Testing The partition analysis method [20] compares a procedure's implementation to its specification, both to verify consistency between the two and to derive test data. Partition analysis selects test data that exercise both a procedure's intended behavior (as described in the specifications) and the structure of its implementation. To accomplish these goals, partition analysis divides or partitions a procedure's domain into subdomains in which all elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. This partition divides the procedure domain into more manageable units. Information related to each subdomain is used to guide in the selection of test data and to verify consistency between the specification and the implementation.

Partition analysis testing is a powerful testing strategy because (1) it integrates several complementary testing criteria; (2) the selected test data exercise the procedure based on both the implementation and the specification. As such, it is one of the few testing strategies to address missing path errors. There are several problems with partition analysis testing that must be addressed. In particular, the proposed testing criteria may result in an excessive number of test points, and most of the them are difficult to implement. Partition testing is not widely practiced and because of the limited experience with this method it is currently not recommended for Class 1E testing.

Domain Testing An input space domain is defined as a set of input data points satisfying a path condition, consisting of a conjunction of predicates along the path. The input space is partitioned into a set of domains. Each domain corresponds to a particular execution path in the program and consists of the input data points that cause the path to be executed. The boundary of each domain is determined by the predicates in the path condition and consists of border segments, where each border segment is the section of the boundary determined by a single simple predicate in the path condition. In domain testing test points are generated for each border segment, which, if processed correctly, determine that both the relational operator and the position of the border are correct [21].

One of the major advantages of domain testing is that, subject to the assumption of a linearly domained program, reliable detection of domain errors requires a reasonable number of test points for a single path. This number of test points grows only linearly with the number of predicates along the path and the number of input variables. The applicability of domain testing to the Class 1E environment is related to that of path testing.

Mutation Analysis Mutation analysis has emerged as providing an approach for evaluation of test data and test methodologies. A mutant program is one in which a statement (or an object instruction) has been changed. A test data set that identifies many mutants can be assumed to also identify native program faults.

Mutation analysis can be viewed as providing a measure of test data quality, and there have been a number of proposals to utilize this analysis in an iterative mode to improve the given test set.

Howden [22] has referred to mutation testing when some of the techniques of mutation analysis are applied to test-set selection. Specially this refers to the construction of tests designed to distinguish between mutant programs that differ by a single mutation transformation.

The underlying assumptions of mutation analysis are still quite controversial and there are serious problems in the implementation. Even a small number of mutation operators can lead to an enormous number of mutant programs. Another problem is the issue of equivalent mutants, and how surviving mutants can be identified as equivalent to the given program or not. It is of interest to Class 1E software only for the comparative evaluation of alternative test approaches.

Symbolic Evaluation In symbolic evaluation, input variables assume symbolic values and output variables are expressed in terms of these symbols. These output variable expressions can then be examined to see if the program is computing the functions intended [23].

Many times the symbolic evaluation approach will show errors that might be difficult to determine using other methods. However, for this approach to be effective, the symbolic expressions should not be too complex; if they are too complex, their usefulness becomes limited. The strategy is of little immediate interest for Class 1E software.

4.2.3 Test Termination Criteria

A fundamental limitation of software testing is that practical programs cannot be tested exhaustively in the sense that the combination of every point in the input domain with every permissible computer state is exercised. The recognition of this limitation has given rise to the frequently quoted dictum "Testing can only show the presence of bugs, but not their absence". Further, any attempt to use statistical methods to establish confidence in the absence of failure mechanisms must be based on equivalence of the failure inducing stresses between the test and use environments. Even where this can be shown to exist, a test interval of five to ten times the desired failure free interval is required to establish confidence levels of 80% to 90% in the achievement of the desired reliability [g]. Thus, to demonstrate 90% confidence in the achievement of a mean-time-between-failures (MTBF) of 1 million hours, a test time of 10 million hours (approximately 11,000 years on a single computer) would be required.

These limitations suggest that the criteria for test termination will for the present be based on judgment rather than objective reasoning. In the following test termination criteria are discussed separately for coverage measures and for statistical measures.

4.2.3.1 Test Termination Criteria Based on Coverage Measures

The following lower level test strategies discussed in the preceding paragraph yield structural coverage measures that are of interest to Class 1E software:

Branch Coverage

Condition Coverage

Path Coverage

Many Class 1E projects will use a combination of functional, structural and statistical testing, and in that event all test cases can be used to achieve the required structural coverage provided that instrumentation to capture the structural properties of the test data sets is in place. The instrumentation referred to above consists of counters (for branch or condition coverage) or token dispensers (for path coverage) that are added to the source code at every decision exit (for condition coverage the counters are attached to the conditions within each decision). The counters are incremented every time a program execution passes through the exit to which the counter is attached. Full branch or condition coverage is achieved when there is a number greater than zero in every counter. Conversely, missing test cases can be identified by the number of counters that read zero.

For path coverage a list of feasible paths (identified by the decision exits that are taken) is constructed, and the actual decision exits taken during a program execution are identified by the token sequence generated during the execution. The path corresponding to that sequence is then marked on the list of feasible paths. Unmarked feasible paths remaining at the conclusion of a test program indicate paths that have not yet been exercised.

The test termination goal in structural testing is typically 100% coverage with a somewhat lower percentage sometimes given as the minimum requirement. When it is decided to accept less than full coverage it must be realized that those test cases will be omitted which present the greatest difficulty in test data selection. These are frequently test cases representing multiple rare conditions, and it can be rationalized that these are extremely unlikely to arise in operation. However, the author's experience on a number of NASA programs has shown that paths containing multiple rare conditions are more likely to contain faults than more frequently accessed paths, possibly because of difficulties the software designer has in recognizing the

actions required by the program to deal with multiple rare conditions. Acceptance of less than full path coverage will therefore increase the possibility of an operational failure under rare conditions.

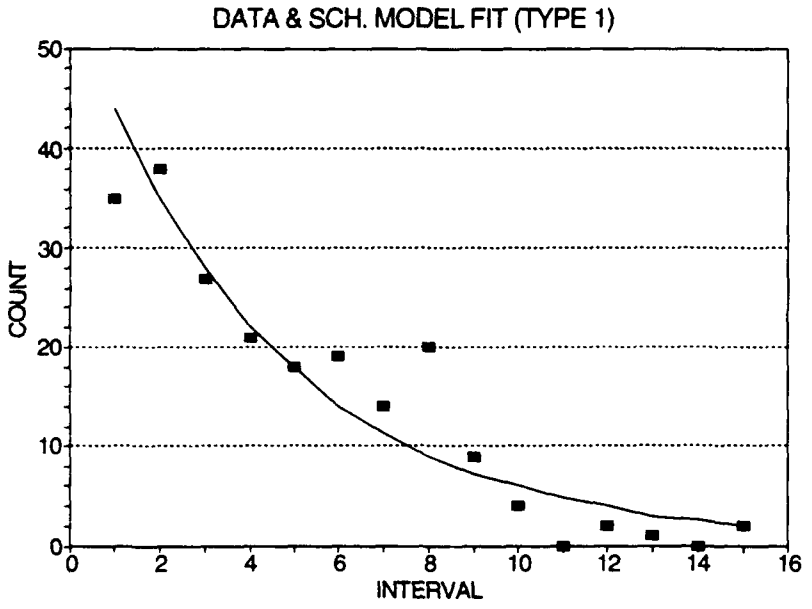
On the other hand, it may turn out to be impossible to generate test cases that access the entire feasible path list, because some structurally feasible paths are semantically infeasible. An example is that an event cannot simultaneously occur during the day shift and at a time earlier than 6 am. Semantically infeasible cases should be purged from the list so that the coverage measure presented for evaluation is assessed against feasibility under both structural and semantic criteria.

4.2.3.2 Test Termination Criteria Based on Statistical Measurements

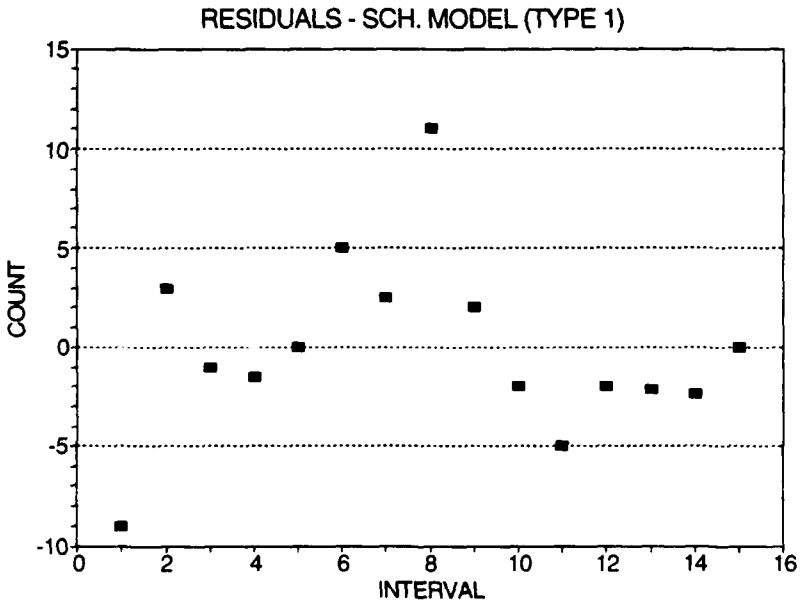
Software reliability measurement and estimation [24] have been used successfully for monitoring the overall test progress and as a test scheduling tool during the initial stages of test. Overall test means the interval from start of integration testing through the end of the acceptance test. Test progress is assessed in terms of the mean execution time between failures (METBF) from one period (week, etc.) to the next. Because faults are removed as they are discovered during the test period, and the time between failures is assumed to be inversely proportional to the remaining fault content, the METBF is expected to increase with test time. Several software reliability growth models have been formulated that permit monitoring whether the time history of METBF on a specific project indicates consistent progress [25]. An example is shown in Figure 4.1. The plot of the residuals (part b) can be used to identify unusual test events, such as the spike in interval 8.

As Class 1E software reaches the acceptance and system test stages the failure rate is expected to be so low that the reliability estimation models discussed above do not yield meaningful results. Nevertheless, the models can provide general guidance for the probable additional test time that will be required to achieve a statistical test termination criterion. In some cases a minimum failure free interval is specified as an acceptance criterion, and the model can then be used to determine when the mean-time-between- failures is likely to reach a multiple, say five times, of the specified failure free interval. The test time forecast by the model can then be used as a predictor of when a failure free test may be run.

Test cases selected for purposes other than statistical testing may meet the defined criteria for statistical testing and can be used to achieve the desired statistical coverage. Acceptance and system testing frequently contain sequences of random data inputs [16].



(a)



(b)

Figure 4.1: Reliability Estimation During Test

The following describes a statistical test technique that is particularly suited to testing of safety systems. The operating range of a safety system can be considered to be the transition region between safe and unsafe operation. While it is necessary to determine that the safety system does not interfere with normal operation, it is also obvious that the primary focus of acceptance testing of the system is the transition region. In statistical testing of a safety system it is therefore appropriate that the input not be randomly selected from the normal operating range of the plant but specifically from the transition range, including both safe and unsafe plant states. A suitable distribution of test inputs that meets these criteria is shown in Figure 4.2. The ordinate in this figure is the probability of the abscissa value being a test input. The parameter types and ranges for choice of random input data can be found in NUREG 1272 "Analysis and Evaluation of Operational Data" [29]. Tailoring the parameters to known plant failure states greatly increases the usefulness of random test for Class 1E software.

Appendix E of IEC Publication 880 provides the following formulas for computing the number of random test cases (n) that have to be executed without failure for statistical confidence c that the failure probability is no greater than P.

$$P \leq \frac{2.99}{n} \text{ for } c = 0.95$$

$$P \leq \frac{4.6}{n} \text{ for } c = 0.99$$

As an example, *30,000* random test cases without failure are required for *0.95* confidence that $P \leq 10^{-4}$.

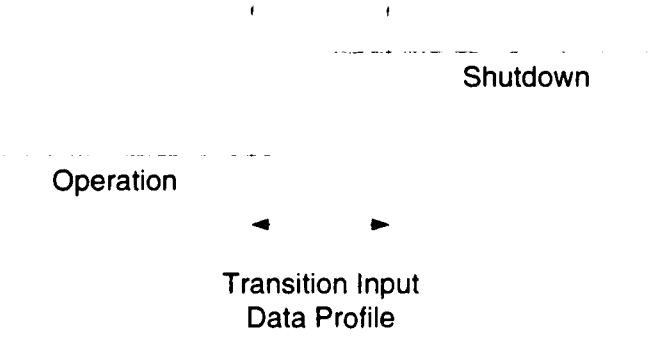


Figure 4.2: Input Profiles for Safety Systems

4.3 Management Considerations for Class 1E Software Test

Under this heading the following topics will be discussed:

Qualifications of test personnel

Applicable standards

Test tools and support software

4.3.1 Qualifications of Test Personnel

It is generally desired that test personnel are (a) unbiased, (b) knowledgeable in the procedural aspects of test and test documentation, and (c) familiar with the product and the application environment and knowledgeable about the sources of errors that have been encountered in the past. The establishment of qualifications in each of these areas is discussed below. It is noted that requirements under (a) and (c) may sometimes conflict in that personnel familiar with the product and the application are likely to have some bias (usually in favor of the product under test). This difficulty is discussed at the end of the following subsection.

The discussion in this section is very deliberately restricted to personnel qualification as contrasted with certification or any other mechanism that establishes a defined class of authorized individuals for the test activity. The reasons for avoiding the latter approach are that (a) there was very little support for it among the organizations contacted in the survey (not restricted to the Class 1E system vendors), (b) there is at present no competent body to establish certification criteria or to administer the certification process, and (c) to the authors' knowledge there is no precedent in the United States for mandatory certification of software test personnel (individual organizations may have an internal certification procedure).

4.3.1.1 Freedom from Bias

Freedom from bias is required for those test activities that are part of the formal evaluation of safety systems and their software, and this usually includes all tests after unit testing. Practically every guidance document for software test, quality assurance or verification recognizes the desirability of organizational independence of the test function from the development function. This can take the form of merely identifying the degree of independence, as in IEEE Standard for Software Quality Assurance (SQA) Plans [r] which requires:

> Organizational dependence or independence of the elements responsible for SQA from those responsible for software development and use shall be clearly described and depicted.

A stronger position is taken in the Quality Assurance Requirements of Computer Software for Nuclear Facilities [d]:

> Software verification and validation activities shall be performed by individuals other than those who designed the software.

The above citation and most of the following ones refer to verification and validation, and these activities are here interpreted as including the formal test phases. Where independent verification is conducted separate from the formal tests a relaxation of the requirements derived here may be appropriate.

The IEEE-ANS Standard 7.4.3.2 [b] is even more specific by stating:

> The verification group shall be organized to be independent of those responsible for the system design.

The latter formulation is consistent with the guidance in IEC Publication 880 [u] which states that the management of the verification team shall be independent of that of design, with a further explanation of this requirement by the following note:

> The requirements for an independent group implies verification either by an individual or an organization which is separate from the individual or organization developing the software. The most appropriate way is to engage a verification team.

The British Ministry of Defence Standard 00-55 [w] applies to procurements by government agencies and raises very stringent requirements for independence of the Safety Auditor (equivalent to the V&V function for safety critical software):

> The Design Authority shall appoint one or more named individuals to act as Independent Safety Auditor from the outset of a project. A separate contract shall be placed to cover the activities of the Independent Safety Auditor.

The guidance for this requirement states:

> The technical and managerial independence of the Independent Safety Auditor from the Design Authority can best be achieved by using an independent company, but an independent division of the prime contractor may be acceptable if adequate technical and managerial independence can be shown at the Director or Board level.

The central requirement underlying most of these formulations is that the funding for the test activities that are part of the V&V process shall not be controlled by the design team. In the survey of safety system vendors that was conducted as part of this effort most companies emphasized that adequate expertise for conducting a comprehensive test required familiarity with the product that could only be found in their organizations. This contradicts the requirement that safety systems and their software be documented in sufficient detail to permit trained individuals outside the vendor organization to evaluate the suitability of the equipment, plan the installation and licensing, and maintain the systems after their installation. Use of an independent organization to conduct or monitor formal test not only avoids the possibility of biased evaluation but also facilitates uniform treatment of the products of multiple vendors. It should therefore be the preferred approach for Class 1E software.

4.3.1.2 Procedural Qualifications

For tests to produce valid results it is necessary to adhere strictly to procedures which are usually only partially defined in the test procedure document. Therefore personnel responsible for the testing of Class 1E software must be trained in the general procedures that govern all testing of critical software as well as in the specific ones that may be imposed by the local management.

Typical of general procedures is the need to keep the software under test, the test environment and the test cases under configuration control. An example of a local procedure is the logging of routine test events, such as the start and finish times of a test segment (logging of non-routine events may be more accurately described as a general requirement).

Under benign circumstances failure to adhere to these procedures will require the repetition of a series of test runs, but in other cases it can invalidate a major test segment. Because of the potential cost and schedule penalties that arise when correct procedures are not followed in the testing of critical software most organizations have training programs in this area. Possibly because of the obvious self-interest of companies in the procedural qualifications of their personnel this aspect has received little attention in the standards and guidance literature. However, adequate familiarity with all procedures affecting the test process is also required for

non-company personnel who may participate in or monitor the test, and for these dependence on self-motivation on the part of the company conducting the test may not be sufficient.

As a minimum, it should therefore be required that general and locally mandated procedures applicable to the conduct of the test be documented and that this documentation be made available to any non-company personnel who participate in or monitor the test. Where formal training in test procedures is provided, this program shall also be open to such non-company personnel.

4.3.1.3 Familiarity with the Product and Application

Familiarity with the product and its application is necessary to interpret and implement test plans and specifications, to assess test results, and to take appropriate action when unforeseen conditions arise during the test. These facts are frequently cited to defend significant participation in test by personnel who developed the software to be tested. As already discussed, the test article should be documented in sufficient detail to permit generally qualified technical personnel (with education and experience equivalent to that of the developers) to conduct or monitor the test, and this negates the need for assigning specific test responsibilities to development personnel (it is not intended to restrict them from supporting test). In this connection a provision of the IEEE-ANS Standard 7.4.3.2 [b] is particularly significant:

> The technical qualifications of the verification team shall be comparable to those of the design team.

Taken together with the following requirement for documentation of verification (including test) established in IEC Publication 880 [u], there is a consistent basis for a technically competent independent test or test monitoring organization:

> The level of detail shall be such that an independent group can execute the verification plan and reach an objective judgement on whether or not the software meets its performance requirements.

It is expected that test staff is familiar with the nature of software faults and has access to the records of failures encountered during development.

4.3.2 Applicable Standards

4.3.2.1 The Role of Standards for Test Methodology

The term *standard* is here interpreted broadly to include documents issued by standards making organizations that may or may not be formally designated as standards. Consideration of standards is important in developing guidelines for testing of Class 1E software and systems for at least the following reasons:

- Standards (established or newly created) provide a common understanding of the test objectives and requirements among the interested parties.
- The use of established standards reduces the cost and schedule for software and system development because of the experience in prior application of methodologies that satisfy the standards.
- Adherence to standards permits conforming products to be widely applied, thus providing motivation for the supplier to invest resources for the development of a superior product; this consideration is particularly important for safety systems that conform to international standards. Standards facilitate interchangeability of competing products, thus providing benefits of potential price or performance competition for the user.

From the large universe of standards that pertain to software and systems test it was necessary to select a small number for discussion in the following sections. The selection was governed by the following considerations:

1. The standard has potential as a reference document for regulatory guidelines
2. While not meeting the criteria for (1) the standard provides definitions or structure that can be used in the formulation of regulatory guidelines
3. The standard is known to the safety systems community and may be used as a basis of comparison with those included under (1) or (2).

Individual standards are grouped below by originating agency. In the discussion of the standards the rationale for assigning it to one of the above three classifications is mentioned.

4.3.2.2 U. S. Military Standards

MIL-STD-882B + Notice 1 [o] System Safety Program Requirements, Mar 1984

The standard contains no specific guidance for test methodology. Definitions and hazard classifications are duplicated in more applicable documents. Designated as (3)

MIL-STD-1521B [h] Technical Reviews and Audits for Systems, Equipments, and Computer Software, Jun 1985

Associates delivery of test documents (test plans, specifications, procedures and reports) with project milestones (reviews and audits). Milestones are based on waterfall model and need to be interpreted for the nuclear safety systems environment. Designated as (2)

MIL-STD-2167A [e] Defense System Software Development, Feb 1988

This standard is aimed at software for mission-critical computers in an environment where software is developed by a contractor for use by military. While not addressing specific test methodologies or test termination criteria, par. 4.3 provides useful guidance for the test environment, organizational independence of test activities, and traceability between requirements and test cases. Designated as (2).

4.3.2.3 Other U. S. Government Standards

NIST SP 500-180 [q] Guide to Software Acceptance, Apr 1990

Section 5 of this document deals with Software Acceptance Testing. It contains no guidance on specific methodologies or test termination criteria but has good checklists for the buyer's (or regulator's) activities and responsibilities in acceptance testing. Designated as (2).

4.3.2.4 Non-Governmental Standards

ANSI/IEEE Std 730-1981 [r] IEEE Standard for Software Quality Assurance Plans

Contains only very general requirements for test, embedded in a verification and validation plan and report (Section 3.4.2.3 and .4). Designated as (3)

ANSI/IEEE-ANS-7.4.3.2-1982 [b] Application Criteria for Programmable Digital Computer Systems in Safety Systems of Nuclear Power Generating Stations, Jul 1982

Section 7 (Verification) contains concise requirements for all verification activities with additional specific requirements for test (see below). It deals with organizational independence and technical qualifications of the verification personnel, and with the documentation of all review and audit procedures. It requires the verification group to provide a test plan that among other items specifies:

- criteria for establishing test cases (par. 7.3.1)
- expected results for each case (7.3.2)
- requirements for testing all logic branches (7.3.3)
- acceptance criteria (7.3.4)
- error reporting and re-testing procedures (7.3.6)

The current standard (discussed above) is widely accepted in the nuclear industry; it is currently undergoing revision with anticipated tightening of some requirements. Designated as (1)

IEEE Std 829-1983 [j] IEEE Standard for Software Test Documentation

This standard is particularly strong in the definition of the software submitted to test, of features to be tested or not tested, and in establishing criteria for acceptance of test results, for suspending test, and for resumption of test after a suspension. Designated as (2)

This guide is intended to supplement IEEE Std. 730 (see above). It requires identification of staffing levels for each QA activity. Designated as (3)

IEEE P1008 [k] Standard for Software Unit Testing, Mar 1985.

Identifies three stages for test case generation:

- Requirements based
- Architecture based (algorithms and data structures)
- Implementation based (code structure)

Designated as (2)

IEEE P1228 [m] Standard for Software Safety Plans, Jul 1991

The current draft contains only broad requirements for test. Designated as (3)

4.3.2.5 International Standards

IEC 880 [u] Software for Computers in the Safety Systems of Nuclear Power Stations, 1986

Test is covered under two separate clauses: for software (clause 6.2.3), and for the integrated hardware/software system (clauses 7.5 through 8.1). In addition, Appendix E contains pertinent tutorial material on verification methods. The requirements are stated very broadly and the only unique feature is found in a discussion of computer system validation (clause 8) where it is *recommended* that tests:

- cover all signal ranges in a fully representative manner
- cover voting and similar logic
- include trip devices in their final configuration
- verify response times and correctness of actions under all failure conditions.

Designated as (2)

IEC 987 [t] Programmed Digital Computers Important to Safety for Nuclear Power Stations, 1989

This standard is targeted at hardware. It refers to IEC 880 par. 7.5 for hardware/software integration testing. Designated as (3).

4.3.2.6 Foreign Standards

MOD 00-55 [w] The Procurement of Safety Critical Software in Defense Equipment

Key test requirements, listed under par. 3.3 (Dynamic Testing) include statement and branch coverage, and testing all loops for 0, 1, and many iterations. Use of a coverage analyzer is specified (it is of course implicit in the coverage requirement). There are stringent requirements for the independence of the test group, including keeping the test cases hidden from the developers. Designated as (2).

MOD 00-56 [x] Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defense Equipment

As part of the preliminary hazards analysis target failure rates are established for each function which can be propagated into test objectives. Designated as (3).

4.3.2.7 Evaluation of Test Standards

Many of the standards address the organizational aspects of software test: independence of the test group and qualifications of test personnel. Comparatively few deal with the technical requirements of test, and none go substantially beyond the complete branch testing specified in ANSI-IEEE-ANS 7.4.3.2. There is thus no basis for preferring another existing standard to the one currently in use in the U. S. nuclear industry.

Desirable provisions from other standards that should be adopted into 7.4.3.2 or else separately imposed as regulatory guidance include:

- use of algorithms and data structures as a source of test cases (IEEE 1008)
- definition of test articles and scope of test (IEEE 829)
- verification of response times (IEC 880)
- traceability between requirements and test cases (MIL-STD-2167A)

In addition, MIL-STD-1521B provides valuable guidance for the scheduling of test documentation, and NIST Special Publication 500-180 identifies the sponsor's or regulator's responsibility during acceptance test. These issues are currently outside the scope of 7.4.3.2, and therefore it may be expedient to treat them in the regulatory guidance document.

While path testing is not currently required in any standard, it is believed that selective application of this methodology is highly desirable, because it provides systematic test coverage for multiple rare conditions.

4.3.3 Software Test Tools

Automated assistance for testing is motivated by both technical and economical considerations. Software test tools can (1) speed up the testing process, (2) reduce the amount of labor required, (3) improve the thoroughness of testing, and (4) make the quality less dependent on skills of the test staff. Automated tools make it possible to achieve a level of thoroughness in the testing process that would be difficult, if not impossible, to accomplish manually. The fourth benefit is of interest to regulatory agencies because personnel qualification criteria are difficult to

implement and enforce. An additional advantage of tool based software testing from the point of view of a regulatory body is that the documentation will be more uniform than that manually generated and thus more easily reviewed. To achieve these benefits the tools must be dependable and accurate. For use in the Class 1E environment it is desirable to concentrate on tools that have been in use for several years and for which good vendor support is available. In general, test tools are less apt to give rise to serious or common mode failures than do development tools, such as compilers.

Against these arguments for tool usage must be weighed the cost of the tools, additional training, and potential requirements for changing the established development and testing processes. For this reason there is frequently resistance to the use of tools or tool abandonment when initial expectations are not met, either due to limitations of the tool or due to inadequate training.

For Class 1E software the benefits from use of tools during testing are expected to outweigh the disadvantages, particularly in the long run.

Test Tools can be categorized according to their functions:

1. Tools supporting complexity measurement
2. Tools supporting syntax and semantics analysis
3. Tools supporting test coverage analysis
4. Tools supporting regression testing
5. Tools supporting test data generation

Some of these categories, such as test coverage analysis, complexity measurement, and syntax and semantics analysis, are quite mature and recommended to be used by Class 1E software vendors. Although the capability of the regression testing tools are limited, they should also be used whenever applicable. Test data selection tools are still in the research stage. Commercially available test case generators are very limited in their capabilities. They can be used for creating some test data if under favorable circumstance.

The Section A.2 describes each of these categories and summarizes the available tools and their capabilities.

4.4 Criteria for Review of Class 1E Software Test Programs

The objective of this section is to provide criteria for the evaluation of test programs for Class 1E software. The professional literature and current standards summarized in earlier sections provide valuable background but do not directly lead to such criteria because they lack detail or address only a portion of the concerns. The selected approach is therefore to list major topics for the evaluation of test programs, to describe the key issues under each topic, and to identify information that will show whether the issues have been adequately addressed.

The major topics covered in this section are:

1. Definition of the application and its environment
2. Testing in the system and software development cycle
3. Management and organization of test activities
4. Applicable test and documentation standards
5. Test methodology and test termination criteria
6. Test environment and test tools
7. Documentation and review of test results

The discussion assumes that the software under test is new and that the test program is being evaluated at the start of the software development with periodic reviews during the development phase. Usually the information required for the last three topics will only be available after major design milestones.

4.4.1 Definition of the Application and its Environment

The key issue is to determine whether there are adequate requirements for deriving a top level test program. The application should be defined by a systems requirements document as well as by a software requirements specification. The systems level document is needed for evaluation of the systems test planning and the software document is needed for evaluation of the acceptance test planning.

In order to support test the systems requirements must identify at least the following:

- Required functions in each mode of operation
- Criticality assignment for required functions
- Prohibited actions or outputs
- Safe states (when required functions are not operative)

It is highly desirable that the systems requirements or a separate hazards analysis identify the maximum allowable failure rates (or probability of failure on demand) for critical functions.

At least the following must be known about the environment of the application:

- Computer types, redundancy provisions, and primary power supplies
- Human-machine interfaces (information displayed to operators and modes of operator intervention)
- Communications architecture (inter-processor and processor to plant)
- Networking facilities
- Operating system(s)
- Hardware and software maintenance facilities and policies

Where the systems test is to be conducted at other than the target installation the above information is required for both the test site and the target site.

At least the following information for planning the acceptance test must be available from the software requirements specification or associated documents:

- Hierarchical description of functions served by the program with ranking of their criticality
- Definition of all variables (range, units, format, accuracy, time dependencies)
- Description of the relations between the variables

- Throughput requirements (average, normal and peak maximum, duration associated with the maxima)
- Other requirements and constraints (e. g. security, fault tolerance)
- Language and development environment
- Identification of any formal methods that are required or may be used in development

Where the system documents define maximum allowable failure rates or failure probability on demand, the propagation of these requirements to the software must also be available.

4.4.2 Testing in the System and Software Development Cycle

The key issues under this heading are the realism of test planning and the adequacy of the resources that will be available for the test program. The schedule of test activities must be known in at least the following detail:

- Listing of all software and system tests with classification of test activities as informal/formal
- Identification of location and responsible organization for each formal test
- Prerequisite documentation for each formal test and expected availability (includes documentation from non-test activities)
- Hardware and software environment for each formal test and expected availability (including documentation)
- Required training of test personnel (schedule, equipment and software to be used, documentation requirements)
- Schedule of formal and informal tests and their relation to project milestones
- Resource estimates (at least in total staff-months)

4.4.3 Management and Organization of Test Activities

The key issues under this heading are the level of management attention to test, the independence of the test organization from the developer, and the qualifications of test personnel. At least the following information is required for each formal test:

- Designation of individual(s) authorized to start, interrupt, and terminate a test program (e.g., integration, acceptance, etc.), a test sequence, and an individual test run.
- Designation of individual(s) authorized to accept and reject results for the test program, test sequence and individual tests.
- Designation of individual(s) who are custodians of the software under test, the support software, test tools, test cases, test data (results and intermediate) and of the hardware environment. A custodian is the lowest management level that can authorize a change in the configuration of an item.
- Designation of individual(s) responsible for the overall test budget, the budget for individual test programs, for establishing staffing levels, and for hiring/firing of any previously designated personnel.
- Organization charts showing the relationships between the individuals designated above and between the test and development organizations.
- Minimum educational and experience requirements for all test staff positions and for all equivalent development staff positions.
- Actual educational and experience qualifications of the designated individuals authorized to accept and reject test results (at any level) and for individuals who approve design at the equivalent level.

4.4.4 Applicable Test and Documentation Standards

The key issues under this heading are that applicable standards are known and are followed. Standards may be applicable because they are (a) invoked in the statement of work or in the specification, (b) derived from invoked standards or specifications, or (c) selected by the developer. In some cases standards are tailored, or only designated portions of a standard are invoked.

At least the following information is required to evaluate whether the test organization is likely to comply with applicable standards:

- List of applicable standards for the conduct of test, where they are invoked, and prior experience with these standards

- List of applicable standards for the article(s) under test, where they are invoked, and prior experience with these standards.

- List of applicable standards for test documentation, where they are invoked, and prior experience with these standards

In all cases the term ''standard'' is meant to include specifications. The listings should indicate whether standards are to be applied in a tailored or restricted manner.

- Means by which compliance with the standard is checked (a) within the organization responsible for furnishing the product or service, and (b) by the quality assurance function.

- Approved, pending and anticipated requests for waivers of standards.

The authors regard par. 7 of IEEE-ANS Standard 7.4.3.2 [26] as a minimally acceptable basis for test, and Appendix C of the Ontario Hydro standard [27] as offering a considerably more comprehensive approach. The approach and overall structure of tests discussed in Section 2 of the present document can be used with both of these standards.

4.4.5 Test Methodology and Test Termination Criteria

The key issues are that the test methodology and termination criteria are consistent with the applicable standard(s) and can be supported by the test environment (see next section).

The evaluation should consider at least the following:

- Selection of a methodology that is documented and for which adequate training can be provided. It is desirable that there be prior experience with the methodology.

- Consistency of the methodology with the applicable standard(s) and other test requirements (schedule, reporting, etc.)

- Selection of test termination criteria consistent with the applicable standard(s), the available resources, particularly personnel and computer time, and the test environment.

- Provisions for random testing in the transition region between safe and unsafe operation where such testing is not already a part of the test termination criteria.

- Availability of test tools to support the required or selected test termination criteria.

The concerns with resource and schedule limitations can be satisfied by means of personnel, computer, and tool loading charts which allow adequate margins for faults in the initially submitted software and for errors in the implementation of the test methodology.

4.4.6 Test Environment and Test Tools

The key issues under this heading are that the test environment and tools are stable, that they are compatible with the software or system under test and the selected methodology, and that they support the efficient collection and analysis of test results. Because of significant progress in the test environment and tools areas there is an inherent conflict between stability and efficiency since a tool that has been in use for a number of years is not likely to be efficient by current standards. While some compromises will have to be made, it is desirable to avoid tools just emerging from research because they are likely to require frequent modification during test which may invalidate previously captured results. It is also undesirable to accept an environment as stable if it requires many manual operations that can be automated by established techniques because the need for the introduction of automation will very likely become apparent during test and may cause disruption of the test process. In general, testing conducted with automated data input and output is more easily analyzed and produces more consistent final results than testing mainly dependent on manual intervention.

At least the following capabilities should be evaluated to determine that the test environment and tools can support an effective test program:

- Availability of full documentation for the test hardware and the test support software (should be available at least three months prior to the due date for the earliest test document).

- Availability of vendor support for hardware and support software (should be on-site or on-line).

- Familiarity of test operations personnel with the hardware and support software.

- Availability of efficient support software for transforming the source program into machine readable code. Consider that this process will have to be repeated every time

the code is changed to correct faults, to improve the operational characteristics of the software, or to accommodate instrumentation provisions of the test tools.

- Compatibility of the process of generating test cases with the configuration control mechanisms for identification and archiving of test cases.

- Compatibility of tool features with the requirements of the applicable test standard and the selected methodology (e. g., a tool designed to measure branch coverage is not suitable where condition coverage is required)

- Availability of detailed diagrams for test data flow during preparation of a test sequence (analysis of source code and instrumentation), during test execution, and for later analysis of test results.

4.4.7 Documentation and Review of Test Results

The major issue under this heading is that documentation must show compliance with requirements, both in the execution of the tests and in the interpretation of test results. A lesser but very frequently encountered issue is inconsistency between the format of the most detailed data and that at successively higher levels of summarization. This arises because reporting of test results is essentially a top-down process, in that the report starts with the overall conclusions (the unit under test meets or does not meet the test criteria) and then substantiates these conclusions at more detailed levels while the data collection results proceeds in the opposite direction, bottom-up. Very detailed planning of data collection is required to overcome this difficulty.

All test documentation should demonstrate at least the following capabilities or attributes:

- Presence of concisely formulated test objectives with reference to governing requirements for the execution of the test and for the software characteristics to be demonstrated.

- Substantiated compliance with the requirements for the document, and, where applicable, for the performance of the test or of associated tasks.

- Description of features selected by the test organization that were not governed by the requirements, e. g., a specific methodology or tool, in sufficient detail to let a reviewer judge their suitability.

- Identification of the individuals and job titles responsible for data collection, data analysis, and report preparation to enable the reviewer to judge compliance with requirements for personnel qualifications.

Preparatory documentation (test plans, specifications and procedures) should demonstrate at least the following:

- Recognition of resource constraints and ability to perform the required tests within these (e. g., by indicating the number of personnel at each skill level that will be required, computer time, etc.)

- Detailed planning for data collection and analysis to provide data in the format required in the test report.

During the conduct of a test some discrepancies will usually be encountered. Not all of these are due to faults in the test article; frequently problems with the test environment or the test procedure are responsible. A creditable test report will clearly describe the discrepancies, indicate how they were resolved, and whether they invalidated any part of the test results obtained prior to their resolution.

Considerations in the evaluation of a test report therefore include:

- Clear recommendations for the acceptance, conditional acceptance, or rejection of the test article with reasons for the recommendations and references to test data that support the reasons.

- Descriptions of discrepancies, whether due to the test article or other circumstances, actions taken to resolve them, and their effect on the test.

- Where there is not an unconditional recommendation for acceptance, the required remedial actions should be identified.

In the review of the test report the key issue is to determine whether the required capabilities for the article under test have been conclusively demonstrated. The reviewer must be convinced that

- The requirements for the test article have been correctly interpreted by the test organization

- The test was conducted under appropriate conditions (in the environment and with the tools specified in the preparatory documents) and by qualified personnel

- The test results are complete, were correctly evaluated against expected results, and were correctly interpreted against the requirements.

4.5 Conclusions and Recommendations

The background material presented in Sections 4.2 and 4.3 has been used to generate guidance for the review of Class 1E software test programs in Section 4.4. The present discussion addresses broader issues that affect test programs for Class 1E software.

The most important limitation encountered in the conduct of test programs for critical software is that testing cannot be exhaustive. The difficulty represented by this limitation increases sharply with program size because the number of possible interactions is an exponential function of size, and the resources available for test can usually not be raised much more than in linear relation with size.

Possible resolutions of this problem are (a) to place restrictions on the size of programs, (b) to accept testing of limited scope and (c) to require a major expansion of the resources allocated to test as programs get larger. The latter is considered a measure of last resort, and only alternatives (a) and (b) will be discussed in detail.

Where the objective is the direct replacement of an analog function with a digital implementation the software requirements are indeed very modest and complete adherence to some acceptable test termination criteria is possible. This approach precludes significant improvements in plant safety and efficiency that are made possible by the application of digital computers. An example of such an improvement is that the set point for a trip meter that in the analog version requires periodic reset by an operator can be made a dynamically computed function of plant state in the digital version. Safety is improved because the resetting is not longer dependent of operator attention, and efficiency is improved because the dynamically computed set point allows the plant to be operated at higher power levels. Therefore alternative (a), restricting the size of the software, carries with it a cost of lost opportunities.

Using the same example to investigate alternative (b), relaxation of the test termination criteria, it is required to trade off the safety impairment (if any) due to this relaxation against the safety improvement due to reduced dependence on operator intervention. The present state of knowledge provides little useful quantitative data for such a trade study but at least points in a potentially fruitful direction: if the program for the trip meter proper can be well isolated from the setpoint calculation, and if the interface between the two functions can be adequately verified, then a linear relation between program size and test resource requirements can be substituted for the previously postulated exponential one. Keeping individual software functions well isolated in design and implementation is also referred to as "loose coupling" and has other desirable properties. Object oriented design and programming provide loose coupling, and the full implications of this for test of critical software needs to be studied.

A complementary approach that will benefit any test program but particularly those for large critical programs is to target the testing at specific software failure modes that can produce hazards. To this end it is necessary to identify

- the software outputs or system level functions where hazards can occur
- the fault types that can produce hazards.

The first requirement can be met through established system safety procedures, such as fault tree analysis or hazardous operations analysis. The second one needs more research, but as mentioned in the body of this report, faulty processing under multiple rare conditions usually escapes detection by conventional test methodologies and is thus a prime suspect for producing hazards. Path testing is a known systematic approach for covering multiple conditions, but because of the potential for requiring an excessive number of test cases it must be used sparingly. Under the specifically targeted conditions advocated here it is both feasible and effective. Further research into this field, which may deserve the title "Testing smarter instead of testing more", is highly recommended. Among the objectives should be the collection and dissemination of data on Class 1E software failures during test, particularly in the later stages, and analysis of these data to identify improvements in both development and test methodologies.

The inherent limitations of test must be recognized, but improvements in development and test methodology hold the promise of containing or shrinking the domain of hazards not likely to be detected by test. Research is recommended in

- The benefits of object oriented design and programming on testability of Class 1E software, and
- Targeted testing for functions and fault types likely to cause hazards.

Chapter 5

Fault Tolerance and Fault Avoidance

5.1 Introduction

5.1.1 Motivation

Fault tolerance and fault avoidance are techniques for reducing the likelihood that a fault will cause a disruption of an important service. The aim of fault avoidance is to prevent or reduce the occurrence of faults, while fault tolerance is directed at dealing with the effects of faults before they become manifest at the system level. The need for fault tolerance for hardware for Class 1E systems is not challenged because the development of faults in hardware, particularly in electronics, is considered as inevitable. Because software is not subject to physical deterioration the need for fault tolerance in this field is not quite as obvious. Is not software development an inherently logical process that, if carried out correctly, should yield a fault-free product? Our findings are that some programs exist in which no faults have been found but that it is impossible to define techniques that give high assurance that a future program will fall into that group. The premise of this chapter is that the need for software fault tolerance arises from the inability to distinguish between programs that are initially free of faults and those that are not, rather than from the need to protect against deterioration of an initially fault-free component (as in hardware).

5.1.2 Structure of this Chapter

Section 5.2 discusses the nature of software failures and provides a basis for understanding fault tolerance as well as fault avoidance. This is followed by individual sections on fault tolerance and fault avoidance. The final section presents conclusions and recommendations for further research.

5.2 The Nature of Software Failures

5.2.1 The Failure Process

The cause of a software failure is a fault in the program, and the effect of the failure is an error, a deviation of the service furnished by the program from the desired service. The failure is an event in the computer when the content of a register transitions to an incorrect value (the value that results in the error). Programs developed by a responsible organization will not produce errors during most executions since they have undergone extensive tests to detect and remove faults responsible for such errors. Therefore the failure is usually due to the presence of an unusual circumstance (data value or computer state) which is called the trigger. The relationship between these concepts is shown in Figure 5.1.

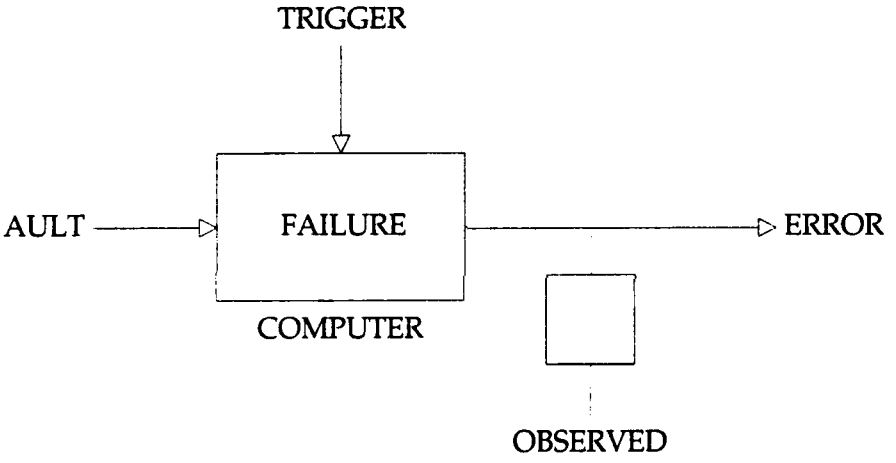


Figure 5.1: Failure Terminology

In a very simple case the fault may be the lack of a divide-by-zero protection, the trigger the occurrence of a zero divisor, the failure is the event of overflow of a register, and the error is the overflow interrupt. In a sophisticated digital system the occurrence of an overflow condition may invoke an exception handler that prevents the propagation of the overflow error and thus avoids or mitigates the failure from appearing at the system level. The need to study the nature of failures at several levels has long been recognized [28]. For software failures at least three levels can usually be identified:

- the logic level, where the error is typically manifest as an incorrect binary value

- the information level, where the error is typically the incorrect value of a variable
- the system level, where the error is typically an undesired action (display, positioning an actuator, printing a message, etc.)

The relation between these is shown in Figure 5.2. No triggers are shown for the information and system levels but these are possible, typically causing disablement of protective measures. In this particular example the existence of faulty data was recognized at the information level (e.g., by violation of embedded assertions) and that caused rejection of the message. Conceivably a further condition could arise to defeat the intent of the assertions, and that condition would then constitute a trigger at the information level. If the anomaly had not been detected at the information level the "lost message" error at the system level could have been transformed into a considerably more severe event. Practically all documented severe malfunctions in critical systems involve multiple triggers and the deliberate disablement of some protective measures.

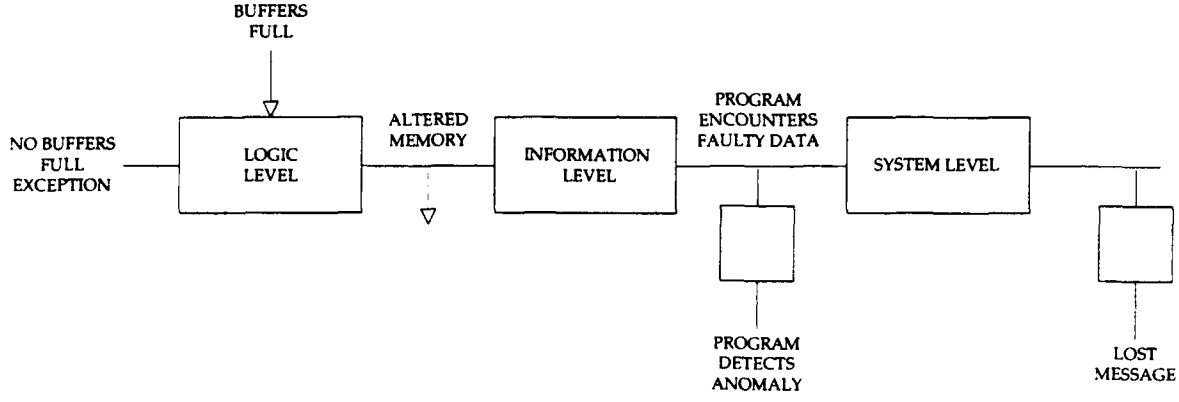


Figure 5.2: Failure Manifestations at Several Levels

The representation of failures in Figures 5.1 and 5.2 shows that the failure probability is a function of:

- the arrival rate of trigger conditions (at all levels)
- the fault content of the software
- the extent and robustness of protective measures at the information and system levels.

The first of these variables is outside the scope of software development and test. Very high quality practices can minimize the number of conditions that act as triggers but they cannot reduce it to zero. It is here assumed that fault tolerance will be required even where sophisticated fault avoidance is practiced.

The importance of trigger conditions in determining the failure probability also motivates continuous study of anomalies in plant operations and computer performance in this environment in order to arrive at a creditable failure prediction model. The view of failures as being a function of both fault content and trigger rates is needed not only for establishing whether a given installation complies with safety criteria but also for quantitative assessment of operational practices. These operational practices can improve or detract from the baseline reliability of a Class 1E system.

5.2.2 Severity of Failures

Fault tolerance is a costly provision, and therefore it should be employed only to protect against faults that can cause severe failures. In the following, several approaches to severity classifications are discussed.

The commonly used severity classifications in the aerospace and defense environment are based on system effects. Examples include:

1. From MIL-STD-1629 (Failure Modes, Effects, and Criticality Analysis) [i]

 Category I Catastrophic --- May cause death or major economic loss

 Category II Critical --- Severe injury, major property damage

 Category III Marginal --- Minor injury, minor property damage

 Category IV Minor --- Not serious enough to cause injury or property damage

2. From British Ministry of Defense Standard 00-56 (Safety Classification of Computers and PES) [x]

 Catastrophic --- Multiple deaths

 Critical --- Single death or multiple severe injuries

 Marginal --- Single severe injury or occupational illness (or multiple minor ones)

Negligible --- At most a single minor injury

Although the specific categories are not aligned, the standards use essentially the same criteria, based on effects usually observable only in an extended system context and after the fact. In MIL-STD-1629 it is recognized that the lethality or the extent of injuries cannot usually be determined a priori, and a method is suggested to translate lower level effects (e.g., complete or partial outage of a computer, into the end effects by means of beta (β) factors which denote the probability that a component level effect will cause a given severity category. In addition, alpha (α) factors can be used to translate an effect below the component level into a component level effect. An example is that a memory address error may cause an inappropriate result with 0.9 probability and computer shut-down with 0.1 probability. In principle this approach may be suitable for severity assessment of Class 1E software and system failures but it will initially be difficult to obtain creditable values for alpha and beta. With the adoption of conservative assumptions about alpha and beta, and the establishment of a reporting system that updates these assumptions, the severity classifications of MIL-STD-1629 offer a workable approach for the Class 1E environment.

There is an implied severity classification in 10CFR50 Appendix E (with reference to NUREG-0654) consisting of the following plant conditions (from most to least severe):

- General Emergency
- Site Area Emergency
- Alert
- Unusual event subject to notification requirements

The criteria for invoking these states are tied to specific observation at the time of the incident (radiation levels, containment pressure) and are therefore not suitable for classifications of failures in protective systems. Another disadvantage is that the Emergency Plans are specific to each site and may therefore be applicable only in the local environment. This precludes their use for a generic severity classification.

An internal document generated by Ontario Hydro mentions four severity classes for software safety:

- Category 1 - Safety critical
- Category 2 - Significant effect on safety

- Category 3 - Some effect on safety
- Category 4 - No effect on safety.

This classification, like that discussed immediately above, is based on the function of the software (to which is associated a worst case effect) rather than on the effect directly. It is not considered more suitable than the preceding ones because it is derived from an internal document and has therefore received much less scrutiny than the accepted standards.

5.2.3 Frequency and Criticality Classifications

The need for fault tolerance is also a function of the (expected) frequency of failures. Thus, a high severity fault with very little likelihood of occurrence will not necessarily require the same degree of protection as an equal (or possibly even lower severity fault) with a high probability of occurrence. Typical expected frequency classifications are: frequent, probable, occasional, remote and unlikely (or improbable).

An example of criticality (or safety integrity) rankings from MoD Std. 00-56 is shown in Table 5.1.

Table 5.1: Criticality Ranking

Failure Probability	Severity			
	Catastrophic	Critical	Marginal	Negligible
Frequent	S4	S4	S3	S2
Probable	S4	S3	S3	S2
Occasional	S3	S3	S2	S2
Remote	S3	S2	S2	S1
Improbable	S2	S2	S1	S1

The criticality assignment involves even more judgment than the severity classification because a further assessment for expected frequency of occurrence is required.

On the other hand, the selection of fault tolerance provisions should not avoid consideration of expected frequency of events that require fault tolerance, particularly where there are creditable data on the occurrence of events such as NUREG-1272 [29]. In the Class 1E environment it is not appropriate to neglect protecting against a variety of faults because it has never been observed. But it is not unreasonable to require higher coverage for events that have been repeatedly observed than for those that have not. This approach is used to evaluate some of the fault tolerant architectures in a later section.

5.2.4 Error Type Classification

The function of protective systems is typically to take a prescribed action when defined conditions arise. The error in the operation of the protective system can be

- Type 1 --- failure to act when the defined conditions have been met
- Type 2 --- action when defined conditions have not been met.

While the primary emphasis in fault tolerance and fault avoidance is to prevent Type 1 errors there must also be safeguards against excessive frequency of Type 2 errors. In general, the probability of a Type 2 error is directly proportional to the number of independent channels that can by themselves initiate a protective function. Requiring agreement among two independent channels before a function is initiated will usually contain Type 2 errors at an acceptable level.

5.3 Fault Tolerance

5.3.1 Architectures for Fault Tolerance

Four architectures for fault tolerance are shown in Figure 5.3 in a specific implementation for software fault tolerance. In all cases the simplest possible example of a technique is shown, and in particular the capability of architectures b through d to use more than two versions of a program or function has not been depicted. The order of presentation is from the generally least effective (retry) to the generally most effective (functional redundancy). Repeating a previously unsuccessful execution can overcome failures due to temporary causes, such as a full queue or a busy communications link. Although the fault tolerance capability of the architecture shown in part a of Figure 5.3 is thus limited, the technique is highly cost effective since it is so easy

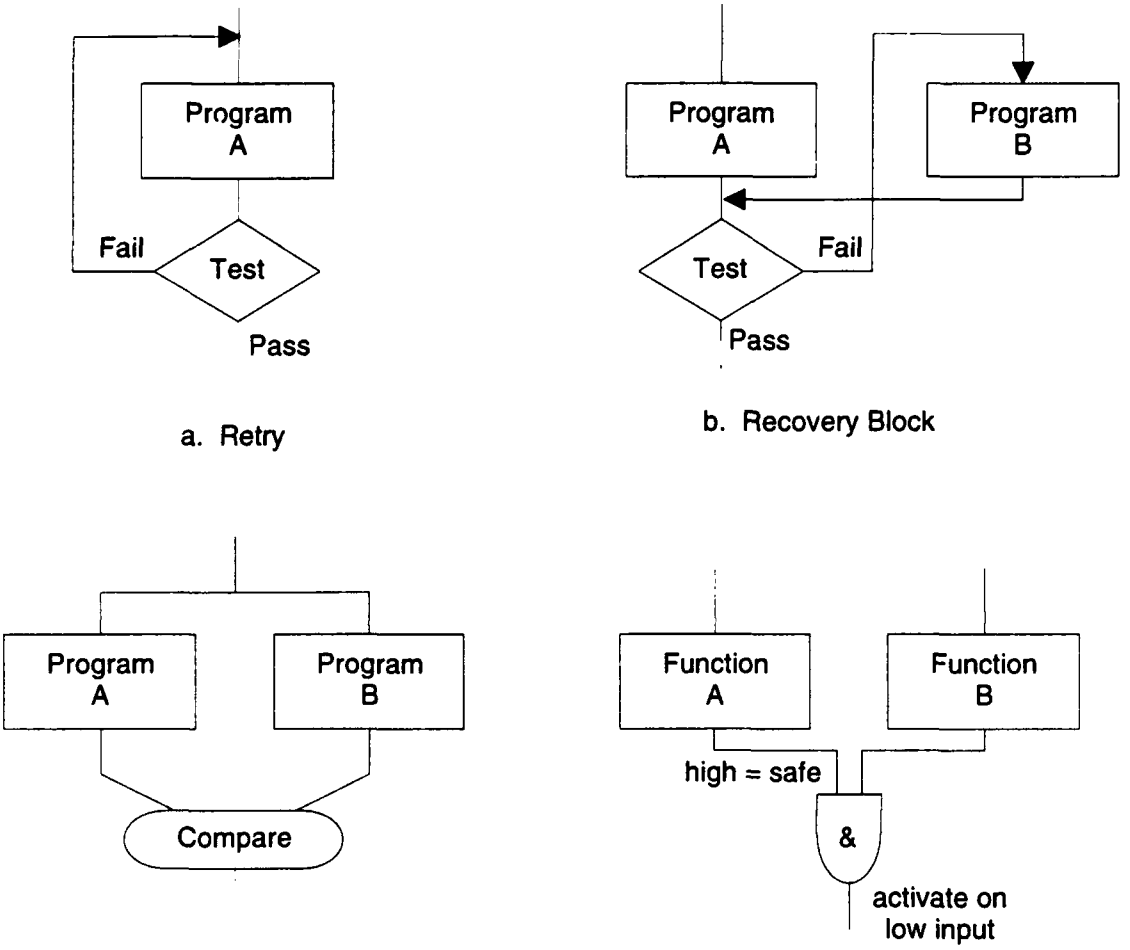


Figure 5.3: Fault Tolerance Architecture

to implement and copes with faults due to temporary exception conditions (such as a blocked communications channel) against which other approaches are not always successful.

In the recovery block approach another program version (which could be functionally redundant) is executed after the first one fails. It can cope with a broader spectrum of causes of failure than the retry but it requires the independent development of two versions and is therefore much more costly. The test block (usually referred to as acceptance test) is common to architectures a and b in Figure 5.3, and this identifies both as dynamic fault tolerance techniques, i. e., those that require a decision to switch from normal execution to an alternate path. The design and implementation of the acceptance test are critical features for the effectiveness of dynamic redundancy. If the test does not detect an error in the execution of the program the intended fault tolerance provisions will not become operational. This form of fault tolerance should therefore be selected only where good criteria for an acceptance test are available. Protective systems for nuclear power plants usually meet this condition because the software models a physical process that is subject to laws of conservation (of mass, momentum, etc.) and of continuity from which effective acceptance tests can be formulated.

The remaining two architectures represent static redundancy because there is no explicit transfer of control after a failure of one of the processes. In the c part of Figure 5.3 two versions of the program run simultaneously, the outputs are compared and under normal conditions they yield identical results. The common result is used to control the safety provisions. Disagreements can be resolved in one of several ways:

- use the most conservative result (the one that activates safety provisions) immediately or after a wait of a given number of cycles
- run a diagnostic to determine if intermediate results are reasonable for both programs, and accept the one that has the most reasonable values
- request operator intervention in resolving the disagreement.

The problems associated with resolving a disagreement can be overcome if a third version is provided, which then permits two-out-of-three voting. This configuration is frequently referred to a triple modular redundancy (TMR) with voting and many publications consider it the paramount form of N-Version programming.

The primary drawback of N-Version programming is that two or more independent but compatible versions have to be generated and executed in parallel. These requirements are much more difficult to meet than those for the recovery block, because there Program B is executed only in case of failure of A, and therefore inefficiencies in execution or in its effect on plant operations are only of minor consequence. In N-Version programming A and B are executed every cycle; they must therefore be compatible in exactly agreeing on the results that are being

compared and in having reasonably similar execution times. These restrictions become even more difficult to meet when a third version is added as in TMR.

The effectiveness of software redundancy (this includes the Recovery Block and N-Version programming) depends on two or more versions of the program being independent so that the possibility of correlated faults is negligible. This independence is very difficult to achieve as several published studies have shown [30, 31, 32]. The major sources of correlated faults are associated with the statement or interpretation of requirements and with the handling of multiple rare conditions. Unfortunately these are also the areas that have given rise to most software problems in recently analyzed programs. Thus software redundancy must be crafted very carefully to provide fault tolerance exactly in the areas where fault tolerance is needed most.

N-Version programming, and particularly the TMR configuration, is difficult to implement on the four-channel hardware redundancy that is employed in many current Class 1E applications. Unless separate hardware units are provided for the comparison or voting, these functions have to be performed on each one of the active computers. This involves multiple inter-computer communications at each phase of the voting process (furnishing the values to be voted on, the results of the vote, and the decision to proceed to the next program segment). Because inter-processor communication runs at a much lower speed than the internal processor bus, it can occupy an appreciable part of the execution time for each cycle. Further impairment of the throughput is due to the delay in waiting for the last one of the independent versions to terminate. There are also problems in allocating the different versions to the hardware channels, and in handling the voting or comparison when a hardware channel is down for maintenance.

Functional redundancy ((d) in Figure 5.3) makes it inherently easier to achieve independence than reliance on software redundancy. Functional redundancy is used here to designate a specific implementation of the concept of functional diversity in which diverse plant protection algorithms operate in parallel such that any one of the functions can activate a given safety mechanism. The broader concept of functional diversity is also applicable to situations where one set of measurements or algorithms is used for an automated system and another one for a display. In the specific application to plant protection systems functional redundancy requires hazard detection by two different physical measurements, such as pressure and temperature, and establishing independent requirements for each measurement channel. The requirements are then implemented in high integrity (but not redundant) software that executes on redundant computers in configurations that are examined in detail in Appendix B in this report. As shown in the figure the outputs of the two functions, non-detection of a hazard through pressure and through temperature measurement, are fed to an AND gate such that absence of a safe indication from either one will cause activation of the protective system. Other output algorithms are possible, particularly delay of activation for a number of cycles while only one of the functions indicates unsafe conditions.

The major hurdle in the implementation of functional redundancy is the identification of independent measurements for the same hazard that can be expected to track over the entire operating range. The operations available in digital processing makes the achievement of this

objective much easier than it is in the analog world. Once independent measurements are identified, the attainment of independence of requirements, specifications, design and code is much easier to verify than it is for the redundant software architectures. Thus, while the implementation proper of functional redundancy may involve costs that are comparable with those of the simplest form of N-Version programming ((c) in Figure 5.3), the overall cost, including analysis and test associated with verification, is expected to be lower.

There is a further motive in pursuing functional redundancy for Class 1E systems in a general sense. A typical quantitative objective for failure on demand that is encountered in the literature is 10^{-6}. Demonstrating attainment of this objective by conventional statistical methods (e.g., associated with a confidence level) is impossible. But if this 10^{-6} objective can be partitioned into 10^{-3} objectives for each of two functionally redundant channels verification by statistical methods may become feasible. It is not intended to overlook the difficulties in verifying complete independence even under these conditions, or the paucity of precedents for using statistical testing in evaluating the reliability of computer programs. But this approach more than any other holds promise of establishing objective quantitative criteria that are compatible with the tenets of probabilistic risk analysis that are followed in other areas of nuclear safety.

5.3.2 Error Detection

Error detection is an essential part of dynamic fault tolerance techniques, and it can improve the effectiveness of static fault tolerance approaches. Error detection is also referred to as self-diagnostics. It is frequently implemented as an assertion embedded in the program, such as

```
if <diagnostic == true>
   then continue
   else invoke error handling
```

A brief discussion of the suitability of error detection techniques for use in Class 1E systems is therefore presented here. The most prominent techniques are:

1. Replication checks.
2. Timing checks.
3. Reversal checks.
4. Coding checks.
5. Reasonableness checks.
6. Structural checks.
7. Computer diagnostic checks.

Each of these is discussed below.

- Replication checks consist of performing the same computation two or more times by different means. They provide one of the most powerful and complete measures for detecting errors not due to a common defect among the replications, but they are among the most expensive in terms of the required resources and are therefore recommended only where other techniques are not applicable.

- Timing checks are a common and limited form of replication checks. If the specification of a component includes timing constraints on the provision of service then a timing check can be provided in the system to determine whether the operation of the component meets those constraints. If the constraints are not met then the timing check can raise a "timeout" exception to indicate the error. Timing checks are easy to implement and are effective.

- Reversal checks are applicable in systems where the relationship between inputs and outputs is one-to-one. A reversal check takes the outputs from a system and calculates what the inputs should have been in order to produce that output --- the calculated inputs can then be compared with the actual inputs to check whether there is an error. An example is to square the results of a square root function. Except in special cases like this example, reversal checks have limited applicability and high resource requirements.

- Coding checks are based on deliberate but limited redundancy in the representation of an object in a system, equivalent to the appended parity bits in arithmetic codes. Within the object, redundant check data is maintained in some fixed relationship with the data representing the value of the object. Errors which result from a corruption of either code or data such that this relationship no longer holds can therefore be detected. Coding checks are efficient but can be applied only in special situations.

- Reasonableness checks for acceptability usually are based on a knowledge of the process served by the system. These checks will test whether the state of various objects in the system is "reasonable," based on the intended usage and purpose of those objects as envisaged by the system designer. A set of common checks for reasonableness are illustrated in the Table 5.2 [33]. Reasonableness checks are widely applicable and efficient.

- Structural checks can be applied to the data structures in a computing system. Checks on structural integrity are aimed at inconsistencies in the data structures and are particularly applicable to complex data structures in which a set of elements is linked together by pointers.

Table 5.2: Examples of Reasonableness Checks

Basis	Implementation	Examples
Range	Declared or expected range of variable	Range of coolant flow rate, index of an array
Physical Laws (Newton's, thermodynamic, electrical)	From know limitation of one variable determine limitations of others	change in coolant flow rate is limited by pump capacity
Regular Series	Quantity must increase (decrease) at regular rate	Numbered time ticks or message numbers transmitted from one process to another
Homogeneous Process	Controlled quantity cannot increase (decrease)	Quantity of fuel remaining in reactor
Allowable Transitions	Constraints must not be violated.	A phone call cannot change from ringing to busy.
Accounting Balance	New balance = old balance + receipts - disbursements	Number of fuel elements
Correlated Processes	Increase in one quantity is usually accomplished by increase (decrease) in another	Steam pressure and turbine output

- Computer diagnostic checks differ from other checks in that they are concerned specifically with checking the behavior of the components from which the system is constructed. A typical diagnostic check involves executing a program with a set of inputs for which the correct output is known. Although primarily useful for detecting hardware faults, this diagnostic is also useful for hardware/software interface problems.

5.3.3 Integration of Fault Tolerance Provisions

Large and complex fault-tolerant systems have experienced unanticipated, failure-inducing interactions of individually well-designed components due to oversights in the integration [34]. Especially difficult to integrate are the various hierarchical fault diagnosis and recovery sequences that support the localized fault tolerance of subsystems and programs that serve fault tolerance of functions (''threads'') that are provided by two or more subsystems.

A second major integration problem is presented by two or more nearly concurrent fault manifestations. The large size and distributed nature of new systems lead to the possibility of two or more independent fault manifestations occurring close in time, most likely because of the previously undetected existence of dormant faults. This in turn will require two or more recovery algorithms to be concurrently active, with the resulting risk of mutual interference, deadlocks, and behavior that is very difficult to estimate.

The preferred way of avoiding those difficulties is to keep the program simple. Distributed or concurrent computing is usually not required in Class 1E systems, where it is proposed the need should be investigated by an independent organization. Integration of fault-tolerant systems requires incremental demonstrations of the capabilities, robustness, completeness, and consistency of all fault tolerance functions, and validation requires operation not for only single, but also for multiple overlapping fault conditions.

5.3.4 Evaluation

Fault tolerance capabilities can be arranged in the order of increasing possibility for common software or computer system failure modes as shown:

1. Two or more functionally diverse programs, each running on separate hardware fault tolerant computers.

2. Two or more functionally diverse programs running on the same group of hardware redundant computers with highly robust system software (note 1)

3. Two or more software diverse programs running on separate hardware fault tolerant computers

4. Two or more software diverse programs running on the same group of hardware fault tolerant computers with highly robust system software

5. A single high integrity program (note 2) running on hardware fault tolerant computers

Note 1 Highly robust system software must be (a) simple, (b) incorporate at least time-out and retry provisions, and (c) have test or operational history that verifies a failure rate of less than a specified fraction per installation-year (suggested: 10^{-6}). It must provide at least the following functions (together with the computer on which it is running): (i) round-robin dispatch of application programs, (ii) limit execution of each application program to a preselected time, and (iii) restrict programs to writing only into specifically assigned memory areas. The simplicity criterion requires that functions beyond those listed be kept to a minimum. Multi-tasking and multi-processing programs do not meet this requirement.

Note 2 A high integrity program is one that either (a) has been verified from specification to code by formal methods, (b) incorporates self-diagnostics for all deviations from expected data values or program transitions, or (c) uses the recovery block (Part b of Figure 5.3) approach for software fault tolerance.

The requirements for fault tolerance will be primarily based on (a) the criticality of the function being protected and (b) the existence of alternative means of furnishing the protection function. These factors can be arranged as shown in the following table to select the minimum acceptable fault tolerance capabilities. *This table is presented to show the methodology; it is expected that discussions with industry and reviews by the regulatory and standards community will be required to establish generally acceptable criteria.* The numerical table entries refer to the fault tolerance classifications discussed above. The criticality of the protected function has been expressed in general terms which may be roughly equated to readings of Table 5.3.

Table 5.3: Minimum Required Fault Tolerance Capability

Defense in Depth for the Protection Function	Criticality of Protected Function			
	Highest	High	Medium	Other
None	1	1	2	3
At least one other means of protection available	2	3	4	5

Regardless of the merits of the fault tolerance provisions, software cannot be considered acceptable for Class 1E applications unless it

- implements validated requirements and a verified software specification derived from these
- was generated in a systematic manner, with verification of each development step against the preceding one, and the developer can document adherence to these provisions
- is coded in a standardized language that supports modern programming and test practices and for which there exist adequate development and test tools
- has undergone testing to verify that it meets requirements and does not cause undesired actions (whether in conflict with expressed requirements or not).

Fault tolerance is intended to compensate for deficiencies, primarily in the specification and design processes, that have not been proven to be preventable by current methodologies. It should not be used as an excuse for any deviation from required practice at any stage of the software development and test process. Development or test methodologies that go considerably beyond the minimum requirements indicated above may qualify the software for a higher capability designation. Examples are that the classification may be advanced to the next lower number (except to capability 1) when at lest two of the following are met:

1. Statistical testing that shows a failure probability of less than 0.001 per installation year at 95% confidence level
2. Path and special values testing of all segments that furnish the protection function

3. Formal verification from specification to code (inclusive), except where this has already been used to qualify for class 5

4. Self-diagnostics for all deviations from expected data values and program structure transitions, except where this has already been used to qualify for class 5.

In spite of the possibility of meeting most of the requirements with software diverse programs it is believed that developers will prefer to investigate functional diversity because of the application flexibility (the individual programs can be applied in various combinations and the functionally diverse pair can be applied with few restrictions).

5.4 Fault Avoidance

Fault avoidance means ensuring that requirements, design and implementation faults are not introduced during system development and construction. Fault avoidance can be achieved by using careful structuring, a reduction of complexity and formal verification. Trusted components refer to the components that are virtually failure-free in the operational context. A voting routine in an n-modular-redundant (NMR) system is an example of such a component. Trusted components in a fault tolerant system design may be particularly important in supporting the fault tolerance.

The means of fault avoidance for safety systems can be divided into two categories that are discussed in the following subsections:

1. Enforcement of good software engineering practice.

2. Use of formal methods.

5.4.1 Enforcement of Good Software Engineering Practice

Effective software engineering practice for fault avoidance includes:

- Avoidance of complexity.

- Use of certified software components.

- Use of certified tools.
- Use of configuration management.
- Use of development standards.

Avoidance of complexity implies:

- Re-use of existing software components and standard systems originally developed by effective software engineering practice, given that these meet current functional, quality and safety standards.
- Isolation of safety-critical functions, so that they are easier to implement, verify and certify.
- Avoidance of practices that may be desirable for non-Class 1E code such as dynamic memory allocation and multi-level interrupts.

A certified component (object) must be well-documented, easy to use and the certification should be to a sufficiently high standard. A certified tool is one that has been determined to be of a particular quality. The certification of a tool will generally be carried out by an independent, often national, body, against national or international standards. Tools are necessary to help developers in the different phases of software development. Use of a certificated tool gives more confidence in the results it produces. Ideally, the tools used in all development phases should be subject to certification, but to date only compilers are regularly subject to certification procedures.

Configuration management aims to ensure the consistency of groups of development deliverables under the inevitable changes. Configuration management prevents the errors that can arise from uncontrolled changes to deliverables. In essence, it requires the recording of the production of every version of every deliverable and of every relationship between different versions of the different deliverables. The resulting records allow the system developer to determine the effect of a change of one deliverable on other deliverables. In particular, systems or subsystems can be reliably modified or re-built from consistent sets of component versions.

Using standard approaches to the software development process enhances software quality. A useful set of standards that covers all aspects of the software development project includes the following areas:

- Project organization.

- Project review and inspections.
- Design methods.
- Programming languages and support software.
- Programming style guides.
- Verification and validation methods.
- Test methods and levels of testing.
- Documentation standards.
- Version control and configuration management.

Such standards ensure a consistent approach to the development. If the standards are properly implemented, the approach should minimize faults by imposing good programming style which should also ease subsequent software maintenance.

5.4.2 Formal Methods

5.4.2.1 Statement of the Problem

The system and software specifications for a digital plant protection system are key determinants for its integrity and performance. Therefore regulatory documents for Class 1E digital systems must be concerned with the methodology used in the formulation of these specifications.

The use of Formal Methods (FM) for the formulation and verification of specifications for digital systems and particularly for software has recently received wide attention in the technical literature and is being mandated or given pronounced preference in the U. K. Ministry of Defense Interim Standard 00-55 and in other proposed (draft) European standards. As yet there are few industrial (contrasted with research and teaching) applications of FM in the U. S. and none in the nuclear industry. Nuclear industry personnel interviewed so far are inexperienced in FM and have indicated no interest in using this methodology.

5.4.2.2 Current Capabilities

Formal methods are presented as "mathematically based techniques for describing system properties," and further "a method is formal if it has a sound mathematical basis, typically given by a formal specification language" [35]. A more recent tutorial article defines formal methods as "that branch of research in the foundations of computer science which deals with modeling and reasoning about (properties of) sequential and distributed systems" [36]. Difficulties arising in applications of formal methods are that they have been misused [37]. Besides the emphasis placed on mathematical foundations and description of properties, emphasis also should be placed on checking the application of a method. Well designed tools, based on the mathematical foundations, are required to bring a formal method into practice. Moreover, there is a distinction between mathematics and formalism. In particular, a formal method should:

1. include a clear statement about the intentions of an application of the method (abstract system design, detailed system design, circuit design, procedural design, etc.) as well as the intended method of checking the results (proofs, testing, reviews, etc.)

2. enable a clear and unambiguous statement of results (systems, hardware, software, etc.)

3. provide effective procedures for checking that the results meet the intentions.

The need of formal methods is not foundational but practical. First, formal methods facilitate making clear and unambiguous statements about systems. Second they exist for the representation of system designs on and manipulation by computer systems.

The following major criteria can be used as a guide to select methods:

- style of presentation
- scope of application
- modularity
- executability
- graphical representability

The style of a specification language can be categorized as either model-oriented or property-oriented [35]. Model-oriented specifications describe behavior in terms of some familiar

structure such as state machines, process or set theory. An advantage is that such models seem to be natural for humans to understand. Property-oriented specifications describe behavior directly in terms of the visible behavior at the interface to the system being specified. Ideally, there is no internal system structure or model implied. An advantage is that ideally there is no implementation bias; that is, the specifications do not give the software engineer any guidelines, or implied constraints, about the internal structure of the system.

The scope of applicability of a specification language is determined by the extent to which the basic concepts in the language directly support the basic concepts in the abstractions being specified. We may also think that the scope of applicability is that range of applications for which intuitive descriptions are most easily produced relative to other languages.

A successful specification method should have the property that component systems can be specified and reasoned about. The component specifications should then be composable to form a specification for a composite system. It should be possible to reason about the composite system using the properties of the components without repeating the derivations for the properties in the context of the larger system. This is called modularity.

Executability of specifications (animation) is useful to test them for reasonable and expected behavior, and to assist in understanding them so that they can be modified and refined as understanding is gained.

The ability to represent a specification graphically can assist in understanding. Model-oriented specifications are generally easier to present and understand because graphical representations are available or possible for many of them.

5.4.2.3 Recommended Regulatory Guidance

In view of the capabilities of current FM and anticipated near term advances, FM shall be identified as a desirable technique for the creation of software specifications for Class 1E Systems. The lack of experience and possible reluctance of industry personnel to use FM immediately suggest that alternative methods shall also be accepted for demonstrating that:

- all possible states for each variable in the specification have been addressed
- all possible sequences of events yield the correct response
- all required actions are permissible for the objects and states that may be encountered
- no defined unintended actions will occur.

These demonstrations are not intended to replace other requirements for insuring the integrity of software design and implementation.

5.4.2.4 Discussion

The increasing interest in FM could be seen in September 1990 when simultaneous special issues of IEEE Computer, Software and Transactions on Software Engineering were devoted to FM topics. In addition, the March 1991 issue of the Transactions on Software Engineering has a special section on Requirements Engineering which focuses on FM. Two instances are known in which FM (in a broad interpretation of that term) have been used on nuclear plant protection software in Europe and in Canada [38], [39], [40] with generally beneficial results. It is thus appropriate to ask what role, if any, FM can play in regulatory guidance for Class 1E digital systems in this country.

While FM can be applied at most phases of the software development process the main emphasis in the current literature is on formal specifications. Design and programming languages furnish a formal approach to those development phases already. Motivation for formal specifications arises from a number of factors of which the most prominent ones are:

1. Faults introduced into the specification are very difficult to find in test and are costly to correct. They tend to persist until much later phases, frequently into operations. FM and the associated manual or, better yet, automated proof of correctness may eliminate most of these faults.

2. At least partially automated methods exist for verifying design against requirements and code against design [41]. Sometimes these uncover inconsistencies in the specification but very seldom do they identify missing or inappropriate requirements. It is hoped that FM will be effective in filling this void.

3. A formal specification promises to be a good starting point for automated design and code generation.

Against these factors that motivate the use of FM must be weighed a number of facts that have in the past prevented the adoption of FM and that can be expected to persist into the future at least as obstacles if not as barriers. It must be remembered that FM have been advocated for over twenty years [42], [43], [44] and that their practical applications are still largely confined to protocols and security kernels.

1. The system specification for a Class 1E system is an interface document between an application domain expert (the plant operator) and a system developer who also has considerable domain expertise. The adoption of FM which are non-domain-specific and emphasize a high degree of abstraction will be regarded as a hindrance to a normal exchange of information. The success of FM in protocols and security kernels may be precisely due to these subjects transcending domain expertise. While the software specification is a step further from the application domain, it is still very much influenced by domain specific knowledge. If a formal software specification is developed from an open language system specification without recourse to domain knowledge there is considerable risk that requirements will be overlooked or misinterpreted. Note also that the lowest program cost and the highest reliability have been found in the "organic" environment (where programmers are part of an organization engaged in the application domain) [45].

2. The formal specification is of necessity derived from a plain language document, either a system specification or a software requirements specification. The required translation is outside of the realm of formal proofs.

3. The system as well as the software specification are dynamic documents. This is specifically recognized in the spiral model of software development [12] but has been a de facto procedure in the nuclear systems field for many years. Specification is an element of design at a higher level, and design involves constant trade-offs of needs against capabilities. The literature on FM does not deal with the adaptation of the formal notation to the need for change and trade-offs. The validity of a formal verification usually extends over a defined scope of the program. Any change within that scope may necessitate repetition of the verification process. A workshop sponsored by the Requirements Engineering Project at SEI recommended an "evolutionary, incremental approach; multi-disciplinary teams; and more up-front requirements work using prototyping." No mention of FM is found in that document [46].

4. The large number of FM languages and approaches that have appeared in recent years leads to the impression that there are significant shortcomings that must be overcome by further development. Although some FM languages have seen more use than others there are no practical guides for selection.

One of the well known FM languages is PAISLey, which has been under development at the Software and Systems Research Center of AT&T Bell Laboratories since 1979. Because good documentation is available in the open literature it is used here as example for the "cultural" factors that must be overcome by FM although the technical advantages of recent FM languages are recognized. The leader of the PAISLey development team recently summarized the results and addressed the obstacles that must be overcome to gain acceptance for FM even in an environment that is very research minded and where exceptionally high levels of software engineering education exist [47]. Of 35 PAISLey applications described in the article, 22 were

the work of the author and were predominantly undertaken for research on the language, 11 originated in academic environments predominantly as classroom examples, and for the remaining two only fragmentary information was supplied but they were probably involved with telephone applications within Bell Labs. One of the author's applications was a Submarine Lightguide system for which it is stated that it became "the official requirements specification document." The author characterizes the acceptance of PAISLey as follows:

> PAISLey has enjoyed a substantial amount of use in education. It is an excellent illustration of many concepts, such as concurrent processes, asynchronous interactions, data flow, functional abstraction, and real-time constraints. ...These attributes make PAISLey a good choice for classroom and laboratory projects.
>
> Although PAISLey has been used on real projects when I participated, there is no autonomous industrial user community. And because I hoped that the experiences of a user community would show cost-effectiveness, I did not plan for any other type of evidence. ...
>
> It is often the case that a little formalization goes a long way. This proved true in the Submarine Lightguide project where the specification of the interface to the database --- a minor piece of the specification --- had disproportionate impact. This piece of the specification is easily readable with almost no training (being no more than signatures of the transactions), but it generated a lively and illuminating review session because this global information had never been systematically recorded and there were many misconceptions.

The "formal" character of FM (being capable of proof, preserving their validity under a defined set of transformations) rests on classifying the requirements of the specific application into concepts at a much higher level of abstraction at which mathematical concepts can be used. As an example the temperature measurement in a vessel can be classified as an observation of the state of the vessel, and the state of the vessel can in turn be classified as a set with a defined number of unique (disjoint) members, such as high temperature, normal temperature, and low temperature. From set theory it can then be "proved" that the vessel cannot be at high temperature state and normal temperature state at the same time. The question whether high, normal and low temperature form the exhaustive membership of the vessel-state set is more realistic. It all depends on the definition of that set. Superficially it may be assumed that the three states form a "naturally" closed set. But what about transition states, start-up and shut-down conditions, etc.? It is in mapping of domain-specific conditions into the limited and abstract constructs of FM that interesting insights can be gained and (possibly) some oversights or misstatements in the specification can be avoided. This is what the last paragraph in the quotation from the PAISLey review article refers to.

Whether significant benefits are realized by employing FM depends to a great extent on whether the domain experts are willing to be trained in the definitions, notation and operations of a specific FM language, and how well the formal specification can subsequently be proven and faultlessly translated into the design and source code. Several problems arise along the way:

1. FM depend on either set theory and prepositional calculus or on advanced graph structures such as Petri nets; some approaches require both. Because the basic symbols of these disciplines are inadequate or highly inefficient for representation of practical specifications they have to be augmented by specialized symbols defined for each language. Examples of the use of special symbols in Z and VDM are shown in Figure 5.4 [35]. These languages are generally regarded as among the most readable ones, and the specification of the symbol table is one of the easiest ones to implement in FM. How much training is required to become proficient in a typical FM language? In the previously quoted work on PAISLey it is stated that roughly five days of training may be sufficient to write a specification. But this is followed by an important reservation "... training time is dramatically affected by factors such as whether students are familiar with ... concurrent processes and functional programming [and] whether classroom examples are familiar ..."

2. With regard to tool support for proof and transformation of programs one of the foremost proponents of FM offers the following observations: "[In the U. S.] the tools-heavy security-driven approach has produced a vibrant research community ... but to date few lessons have been carried over to other ... applications ... where more attention must be paid to cost. ... By contrast, the variants of the European-originated Z and VDM methods can be more readily ... used with little or no tool support ..." [48]. This is also evident in a description of an apparently successful application of VDM by Rolls-Royce and Associates to a microprocessor based protection system [49]. As was pointed out in our interview with the FM specialists at SEI, the primary benefit of their approach based on Z is in forcing the domain expert into formal thinking about the specification, not in proving it, and the same appears also true of PAISLey. However, this also implies that the activities downstream of the specification phase are essentially conventional, must be followed by conventional verification and test, and are subject to the same risk as exists at present.

3. Decisions that should be deferred until design may be impacted by the formal specification or, if the constraint is not recognized, the decisions may invalidate the specification. Examples of this type arise in the partitioning of the overall software into programs and of the programs into modules, in arriving at a sequence of executions, and allocation of programs to processors. Even where the main program runs in a continuous loop on a dedicated processor, as is the case in most Class 1E systems, there are program interfaces with supervisory, diagnostic, and hardware test processors that require very careful consideration under these constraints. Separation of plant operation functions from display handlers, record generation, etc. is highly desirable for safe operation of the

Cut and Paste

Figure 5.4: Examples of Formal Methods Languages

system but may not be attainable if it is not recognized in the FM used for the specification.

It is therefore concluded that regulatory guidance may encourage the use of FM but should not mandate it.

5.5 Conclusions and Recommendations

The preceding sections of this chapter have identified promising approaches to both fault avoidance and fault tolerance. While significant strides have been made in systematic approaches to fault avoidance they are still short of demonstrated ability to assure fault free performance. For applications in Class 1E software it is therefore necessary to provide fault tolerance until conclusive proof of fault free performance by other means can be obtained.

Demonstration projects for established techniques of software fault tolerance, primarily the recovery block and n-version programming, have shown some flaws in their performance. In addition, even if complete ability to tolerate software faults were shown, there would still be concern about the completeness and accuracy of the requirements to which all versions were designed. For these reasons functional diversity is recommended as the preferred approach to fault tolerance. Before discussing this selection further, it must be acknowledged that the failure probabilities associated with the software fault tolerance techniques are several orders of magnitude smaller than those of non-fault tolerant software. Thus there is reason to employ software fault tolerance in selected areas as recommended in earlier sections.

By functional diversity is meant the assessment of the plant state by a number of independent variables, such as temperature, pressure, and neutron flux. The employment of several variables, each with its specific algorithm for assessment of the plant state, provides significantly higher assurance against omission of a critical requirement, or misinterpretation of a requirement in the formulation of the software specification, than is possible in a methodology based on software diversity alone. The functional diversity described here also inherently enforces software diversity.

In some applications functional diversity is provided outside the scope of the software effort, such as in a core monitor in an environment where steam pressure and coolant temperature are used in other plant protection systems. In these situations software diversity, or possibly even fault avoidance techniques, may provide adequate reliability.

These conclusions are based on qualitative arguments, and this is also the case with most of the material presented in the body of this chapter. However, significant progress can be achieved by transitioning to quantitative techniques based on specific values for the maximum

acceptable failure on demand of a protection system. This will necessarily involve an assessment of other protective measures that may act against the same hazard and of the probability of events that pose the demand on the protection systems. These are issues of public policy that are not addressed here. But quantitative requirements will allow more cost-effective software techniques and more objective regulatory review to be employed. Research in this area is therefore recommended.

Appendix A

CASE Tools Supporting Class 1E Software Process

Much of the CASE tools information used in this study was obtained from USENET News Articles in news groups such as "comp.lang.ada" and "comp.std.ada," etc.

An evaluation of the tools was not possible within the scope of this effort. However, some of the tools have been evaluated by the USAF Software Technology Report Center at Hill Air Force Base, Utah as part of the *Upper CASE Tools Evaluation Project.* Reports of these evaluations can be obtained from:

Software Technology Support Center
Attn: Customer Service
Hill AFB, UT 84056

A.1 Checklists of CASE Tools for the Design and Development Environment

A.1.1 CASE Tool Features

This section discusses the functional features of CASE tools. The features are used to assess the capabilities of the CASE tools in the design and development environment for class 1E applications. The representative CASE tool functionalities can be classified into eight categories: information capture, methodology support, model analysis, requirements tracing, data repository and documentation. The classified CASE tool features in each category are shown below. This

scheme is used in later sections to identify the features offered by commercial tools and their applicability to the design of Class 1E software as exemplified in the Ontario Hydro standard [a].

A. **Information Capture**

1. system function descriptions
2. data descriptions of system functions interfaces
3. data descriptions of system input/output device interfaces
4. system logical behavior
5. system timing behavior
6. hardware/software context
7. software process control
8. project information
9. missing information prompts

B. **Methodology Support**

1. structured analysis
2. structured design
3. object-oriented design
4. object-oriented analysis
5. object interaction diagrams or structure graphs
6. entity relationship modeling
7. ADARTS
8. Petri Nets
9. statecharts

10. axiomatic specification

11. data flow diagrams

12. block diagrams

13. control flow diagrams

14. state transition diagrams

15. Petri Net diagrams

16. structure charts

17. flow charts

18. object hierarchy or tree diagrams

19. hierarchical diagram organizations

C. **Model Analysis**

1. consistency checking

2. completeness checking

3. man/machine interface analysis

4. behavior analysis

5. scenario-based analysis

6. exhaustive analysis

D. **Requirement Tracing**

1. multiple requirements baselines

2. tracing of system requirements to software requirements

3. tracing of system design specifications to software requirements

4. tracing of requirements to software design

5. tracing of requirements to source code

6. tracing of requirements to software test

E. **Data Repository**

1. initial data input to the database

2. use a centralized database

3. support access control

4. contain test description and procedures

5. support interactive cross-referencing

6. history reports generated

7. difference reports generated

8. multiple baseline revision/versions

9. reconstruction of previous baselines

10. changes implemented across configurations

F. **Documentation**

1. automatic generation of documentation

2. support documentation standards

3. requirement documents

4. design specification

The CASE tools under this study are: CASE-1, CASE-2, CASE-3, CASE-4, CASE-5, CASE-6, CASE-7 and CASE-8. Table A.1 shows the programming languages the tools support and the host operating systems on which the tools can run. CASE-2 addressed here includes the language modules for Ada, C and C++. Table A.36 on P. 146 lists tool names associated with each code.

Our study is based on the information from the evaluation of CASE tool functional capabilities done by Software Technology Support Center, Hill Air Force [50] and Internet News

Articles. Appendices A.1.2, A.1.3, A.1.4, and A.1.5 provide the CASE tools checklists for Design Input Documentation, Software Requirements Specification, Software Design Documentation, Code, respectively. The checklists are based on the functional features.

Table A.1: Programming Language and Host System Information

	Programming Languages			Host Systems			
TOOLS	C	C++	Ada	DOS	Mac	Unix	VMS
CASE-1	✓		✓	✓		✓	✓
CASE-2	✓	✓	✓	✓	✓	✓	✓
CASE-3	✓	✓	✓	✓	✓	✓	✓
CASE-4	✓	✓	✓	✓		✓	✓
CASE-5	✓	✓	✓			✓	✓
CASE-6	✓		✓			✓	✓
CASE-7	✓	✓	✓			✓	✓
CASE-8	✓	✓	✓	✓		✓	✓

A.1.2 CASE Tools Checklist for the Design Input Documentation

The following list illustrates the tool capabilities supporting requirements for the Design Input Documentation (DID). Each of the requirements defined in the Ontario Hydro document (Standard for Software Engineering of Safety Critical Software) is shown and followed by the CASE tools capabilities supporting them (shown in italics).

a. Partition the system so that the safety critical software subsystem is isolated from other subsystems and software functionality not associated with meeting the minimum performance specification of the special safety system is minimized. --- *structure charts* (B.16), *data flow diagram* (B.11).

b. Define all functional, performance, safety, reliability, and maintainability requirements of the subsystem, clearly identifying the safety requirements. --- *requirement documents* (F.3).

c. Identify each computer within the computer system and includes a description or reference to the computers' characteristics, such as memory capacity, instruction sets, speeds, and input/output registers. --- *hardware/software context* (A.6).

d. Define all details of the interfaces with external inputs and outputs. --- *data descriptions of system I/O device interface* (A.3).

e. Define all accuracy requirements and tolerance. --- *requirements documents* (F.3).

f. Define all failure modes, and the appropriate response to them and identifies any degraded operation modes required. --- *requirements documents* (F.3).

g. Define any constraints placed on the design options --- *design specifications* (F.4).

h. Factor in relevant experience from previous developed systems. --- *no supporting feature among the tools under study.*

i. Limit and localize the use of complex calculations upon which safety critical decisions depend. --- *structure chart* (B.16), *data flow diagram* (B.11).

j. Contain no requirements that are in direct conflict with each other. --- *interactive cross-referencing* (E.5), *consistency checking* (C.1).

k. Provide a clear definition of terms. --- *no supporting capability among the tools under study.*

l. Define anticipated changes and enhancements to the system. --- *project information* (A.8).

m. Define each requirement uniquely and completely in one location to prevent inconsistent updates and to facilitate easy referencing by subsequent documents and verification processes. --- *support access control* (E.3), *use a centralized database* (E.2).

n. Contain or reference a revision history. --- *history reports generated* (E.6), *difference reports generated* (E.7).

Table A.2 presents the summary of the capabilities of the CASE tools in our study with respect to the requirements for the DID shown above. The numbers listed in the columns "a" through "n" represent the capability of the corresponding tools supporting the requirements indicated by the letters. For example, CASE-6 has two types of features supporting the requirement "a," namely, structure charts and data flow diagrams. These two features enable a software engineer to analyze the interactions between subsystems so that they can subsequently partition the system with safety. The numbers in the column marked "total" represent the overall extent of capabilities supporting DID.

- Tool(s) offering best support: CASE-2, CASE-8
- Two tool combination(s) offering best support: CASE-2/CASE-8
- Requirements not supported by any tool: h (factor in experience), k (clear definition of terms)

Table A.2: Tool Features in Support of Design Input Documentation

	Ontario Hydro Requirements														
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	
TOOLS	Number of Features in Support														TOTAL
CASE-1	1	0	1	1	0	0	0	0	1	1	0	1	2	0	8
CASE-2	2	1	1	1	1	1	1	0	2	1	0	1	2	2	16
CASE-3	2	1	1	1	1	1	1	0	2	2	0	0	2	0	14
CASE-4	2	1	0	1	1	1	1	0	2	1	0	1	0	1	12
CASE-5	2	1	1	1	1	1	1	0	2	2	0	1	2	0	15
CASE-6	2	1	0	0	1	1	1	0	2	2	0	1	2	2	15
CASE-7	1	1	0	1	1	1	1	0	1	1	0	1	2	2	13
CASE-8	2	1	1	1	1	1	1	0	2	2	0	1	2	1	16

A.1.3 CASE Tools Checklist for the Software Requirements Specification

This section presents the evaluation of the CASE tools with respect to their capabilities supporting the software requirements specification (SRS). The tools are investigated by contrasting the detailed acceptance criteria for SRS to tools' features. The results are shown in Table A.3 to Table A.9.[2] Each table corresponds to one of the criteria categories, namely, completeness, correctness, consistency, verifiability, modifiability, traceability, understandability and robustness. The first column of each table lists the names of the tools. Each row itemizes the features of the corresponding tool supporting the criteria in a specific category. For example, criteria for SRS's modifiability is defined as "define each unique requirement once to prevent

[2]The absence of table corresponding to the criteria of verifiability for SRS is due to the fact that the CASE tools under this study have no feature to support the criteria.

inconsistent update." Accordingly, in Table A.6, features such as "use centralized database" and "support access control" are listed because they are the means of enforcing consistency. That is, the use of a centralized database makes the concurrency control easier and access control assures that all the updates are done by authorized personnel. Table A.10 summaries the results, in which each figure represents the number of tool features that support a specific attribute of SRS.

Table A.3: Tool Features in Supporting Completeness of SRS

Tools	Petri Nets (B.8)	Statecharts (B.9)	Completeness (C.2)	Scenario (C.5)
CASE-1			✓	
CASE-2	✓	✓	✓	✓
CASE-3		✓	✓	
CASE-4			✓	
CASE-5		✓	✓	
CASE-6			✓	
CASE-7			✓	
CASE-8	✓		✓	✓

- Tool(s) offering best support: CASE-2
- Two tool combination(s) offering best support: N/A

Table A.4: Tool Features in Supporting Correctness of SRS

Tools	ADARTS (B.7)	Petri Nets (B.8)	Statecharts (B.9)	Scenario (C.5)	Missed Information (A.9)	Cross-referencing (E.5)
CASE-1					✓	
CASE-2	✓	✓	✓	✓	✓	
CASE-3			✓		✓	✓
CASE-4						
CASE-5			✓		✓	✓
CASE-6					✓	✓
CASE-7					✓	
CASE-8		✓		✓	✓	

- Tool(s) offering best support: CASE-2
- Two tool combination(s) offering best support: CASE-2/CASE-5

Table A.5: Tool Features in Supporting Consistency of SRS

Tools	Petri Nets (B.8)	Statecharts (B.9)	Completeness (C.1)	Scenario (C.5)
CASE-1			✓	
CASE-2	✓	✓	✓	✓
CASE-3		✓	✓	
CASE-4			✓	
CASE-5		✓	✓	
CASE-6			✓	
CASE-7			✓	
CASE-8	✓		✓	✓

- Tool(s) offering best support: CASE-2
- Two tool combination(s) offering best support: N/A

Table A.6: Tool Features in Supporting Modifiability of SRS

Tools	Revision (E.8)	Reconstruction (E.9)	Difference (E.7)	History (E.6)	Database (E.2)	Access (E.3)	Configuration (E.10)
CASE-1					✓	✓	
CASE-2	✓	✓	✓	✓	✓	✓	
CASE-3					✓	✓	
CASE-4	✓						✓
CASE-5					✓	✓	
CASE-6	✓	✓	✓	✓	✓	✓	
CASE-7	✓	✓	✓	✓	✓	✓	✓
CASE-8	✓	✓	✓	✓	✓	✓	✓

- Tool(s) offering best support: CASE-7, CASE-8
- Two tool combination(s) offering best support: N/A

Table A.7: Tool Features in Supporting Traceability of SRS

Tools	Spec to Req. (D.3)	History (E.6)	Cross Referencing (E.5)
CASE-1	✓		
CASE-2	✓	✓	
CASE-3	✓		✓
CASE-4	✓		
CASE-5	✓		✓
CASE-6		✓	✓
CASE-7	✓	✓	
CASE-8	✓		✓

- Tool(s) offering best support: All except CASE-1, CASE-4
- Two tool combination(s) offering best support: CASE-2/CASE-8, etc.

Table A.8: Tool Features in Supporting Understandability of SRS

Tools	Data flow Diagram (B.11)	Block Diagram (B.12)	Flow Chart (B.17)	System Description (A.1)	Interface Description (A.2)	I/O Description (A.3)	Structure Charts (B.16)
CASE-1				✓	✓	✓	✓
CASE-2	✓	✓	✓	✓	✓	✓	✓
CASE-3	✓			✓	✓	✓	✓
CASE-4	✓	✓		✓	✓	✓	✓
CASE-5	✓	✓	✓	✓	✓	✓	✓
CASE-6	✓			✓	✓		✓
CASE-7	✓			✓	✓	✓	
CASE-8	✓	✓	✓	✓	✓	✓	✓

- Tool(s) offering best support: CASE-2, CASE-8
- Two tool combination(s) offering best support: N/A

Table A.9: Tool Features in Supporting Robustness of SRS

Tools	Scenario-Based Analysis (C.5)
CASE-1	
CASE-2	✓
CASE-3	
CASE-4	
CASE-5	
CASE-6	
CASE-7	
CASE-8	✓

- Tool(s) offering best support: CASE-2, CASE-8
- Two tool combination(s) offering best support: N/A

Table A.10: Tools Support SRS

Tool	completeness	correctness	consistency	verifiability	modifiability	traceability	understandability	robustness	Total
CASE-1	1	1	1	0	2	1	4	0	10
CASE-2	4	5	4	0	6	2	7	1	29
CASE-3	2	3	2	0	2	2	5	0	16
CASE-4	1	0	1	0	2	1	6	0	11
CASE-5	2	3	2	0	2	2	7	0	18
CASE-6	1	2	1	0	6	2	4	0	16
CASE-7	1	1	1	0	7	2	4	0	16
CASE-8	3	3	3	0	7	2	7	1	26

- Tool(s) offering best support: CASE-2, CASE-8
- Two tool combination(s) offering best support: N/A
- Requirements not supported by any tool: Verifiability

A.1.4 CASE Tools Checklist for the Software Design Description

This section presents the evaluation of the CASE tools with respect to their capabilities supporting the Software Design Description (SDD). The tools are investigated by contrasting the detailed acceptance criteria for SDD to tools' features. The results are shown in Table A.11 to Table A.19. Each table corresponds to one of the criteria categories, namely, completeness, correctness, predictability and robustness, consistency, verifiability, modifiability, traceability, modularity, and understandability. The first column of each table lists the names of the tools. Each row itemizes the features of the corresponding tool supporting the criteria in a specific category. For example, criteria for SDD's predictability and robustness include that software system provides required response to all identified error conditions. Accordingly, in Table A.13, features such as "scenario-based analysis," "system-timing behavior" and "exhaustive model analysis" are listed. Table A.20 summaries the results, in which each figure represents the number of tool features that support a specific attribute of SDD.

Table A.11: Tool Features in Supporting Completeness of SDD

Tools	Petri Nets (B.8)	Statecharts (B.9)	Completeness (C.2)	Scenario (C.5)	Object hierarchy (B.18)
CASE-1			✓		✓
CASE-2	✓	✓	✓	✓	✓
CASE-3		✓	✓		✓
CASE-4			✓		✓
CASE-5		✓	✓		✓
CASE-6			✓		
CASE-7			✓		✓
CASE-8	✓		✓	✓	

- Tool(s) offering best support: CASE-2
- Two tool combination(s) offering best support: N/A

Table A.12: Tool Features in Supporting Correctness of SDD

Tools	ADARTS (Alphamethod.7)	Petri Nets (B.8)	Statecharts (B.9)	Scenario Analysis (C.5)	Missing Info (A.9)	Cross Referencing (E.5)	Exhaustive Analysis (C.6)	Function Interface (A.2)
CASE-1					✓			✓
CASE-2	✓	✓	✓	✓	✓			✓
CASE-3			✓		✓	✓		✓
CASE-4								✓
CASE-5			✓		✓	✓		✓
CASE-6					✓	✓		✓
CASE-7					✓			✓
CASE-8		✓		✓	✓	✓	✓	✓

- Tool(s) offering best support: CASE-2, CASE-8
- Two tool combination(s) offering best support: CASE-2/CASE-8

Table A.13: Tool Features in Supporting Robustness of SDD

Tools	Scenario-Based	Logical	Timing	Exhaustive
CASE-1		✓	✓	
CASE-2	✓	✓	✓	
CASE-3				
CASE-4				
CASE-5		✓	✓	
CASE-6				
CASE-7		✓	✓	
CASE-8	✓	✓	✓	✓

- Tool(s) offering best support: CASE-8
- Two tool combination(s) offering best support: N/A

Table A.14: Tool Features in Supporting Consistency of SDD

Tools	Consistency Checking
CASE-1	✓
CASE-2	✓
CASE-3	✓
CASE-4	✓
CASE-5	✓
CASE-6	✓
CASE-7	✓
CASE-8	✓

- Tool(s) offering best support: All
- Two tool combination(s) offering best support: N/A

Table A.15: Tool Features in Supporting Verifiability of SDD

Tools	Petri Nets (B.8)	Petri Net Diagram (B.15)	Logical Behavior (A.4)	Timing Behavior (A.5)	Exhaustive Analysis (C.6)
CASE-1				✓	
CASE-2	✓	✓	✓	✓	
CASE-3					
CASE-4					
CASE-5			✓	✓	
CASE-6					
CASE-7			✓	✓	
CASE-8	✓	✓	✓	✓	✓

- Tool(s) offering best support: CASE-8
- Two tool combination(s) offering best support: N/A

Table A.16: Tool Features in Supporting Modifiability of SDD

Tools	Revision (E.8)	Reconstruction (E.9)	Difference (E.7)	History (E.6)	Database (E.2)	Access (E.3)	Configuration (E.10)
CASE-1					✓	✓	
CASE-2	✓	✓	✓	✓	✓	✓	
CASE-3					✓	✓	
CASE-4	✓						✓
CASE-5					✓	✓	
CASE-6	✓	✓	✓	✓	✓	✓	
CASE-7	✓	✓	✓	✓	✓	✓	✓
CASE-8	✓	✓	✓	✓	✓	✓	✓

- Tool(s) offering best support: CASE-7, CASE-8
- Two tool combination(s) offering best support: N/A

Table A.17: Tool Features in Supporting Traceability of SDD

Tools	Req. to Design	History	Cross Referencing
CASE-1	✓		
CASE-2	✓	✓	
CASE-3	✓		✓
CASE-4	✓		
CASE-5	✓		✓
CASE-6		✓	✓
CASE-7	✓	✓	
CASE-8	✓		✓

- Tool(s) offering best support: CASE-2, CASE-8, etc.
- Two tool combination(s) offering best support: CASE-2/CASE-8, etc.

Table A.18: Tool Features in Supporting Modularity of SDD

Tools	Structured	Structure	OOD	Hierarchical	Object
CASE-1	✓	✓	✓	✓	✓
CASE-2	✓	✓	✓	✓	✓
CASE-3	✓	✓	✓	✓	✓
CASE-4	✓	✓	✓	✓	
CASE-5	✓	✓	✓	✓	✓
CASE-6		✓			
CASE-7	✓			✓	
CASE-8	✓	✓	✓	✓	✓

- Tool(s) offering best support: CASE-2, CASE-8, CASE-1, CASE-3, CASE-5
- Two tool combination(s) offering best support: N/A

Table A.19: Tool Features in Supporting Understandability of SDD

Tools	Data flow Diagram (B.11)	Block Diagram (B.12)	Flow Chart (B.17)	Structure Charts (B.16)	Ctrl-flow Diagrams (B.13)	ER Modeling (B.6)
CASE-1				✓		
CASE-2	✓	✓	✓	✓	✓	✓
CASE-3	✓			✓	✓	✓
CASE-4	✓			✓	✓	
CASE-5	✓	✓	✓	✓	✓	✓
CASE-6	✓			✓		
CASE-7	✓				✓	
CASE-8	✓	✓	✓	✓	✓	✓

- Tool(s) offering best support: CASE-2, CASE-8, CASE-5
- Two tool combination(s) offering best support: N/A

A.1.5 CASE Tools Checklist for the Code

This section presents the evaluation of the CASE tools with respect to their capabilities supporting the Code. The tools are investigated by contrasting the detailed acceptance criteria for Code to tools' features. The results are shown in Table A.21 to Table A.29. Each table corresponds to one of the criteria categories, namely, completeness, correctness, predictability and robustness, consistency, structuredness, verifiability, modifiability, traceability, and understandability. The first column of each table lists the names of the tools. Each row itemizes the features of the corresponding tool supporting the criteria in a specific category. For example, criteria for Code's understandability include that the code must be formatted to enhance readability. Accordingly, in Table A.29, features such as "support documentation standard" and "automatic generation of documentation" are listed. Table A.30 summaries the results, in which each figure represents the number of tool features that support a specific attribute of Code.

Table A.20: Tools Support SDD

Tool	completeness	correctness	predictability robustness	consistency	verifiability	modifiability	traceability	Structuredness modularity	understandability	Total
CASE-1	2	2	2	1	1	2	1	5	1	17
CASE-2	5	6	3	1	4	6	2	5	6	38
CASE-3	3	4	0	1	0	2	2	5	4	21
CASE-4	2	1	0	1	0	2	1	4	3	14
CASE-5	3	4	2	1	2	2	2	5	6	27
CASE-6	1	3	0	1	0	6	2	1	2	16
CASE-7	2	2	2	1	2	7	2	2	2	22
CASE-8	3	6	4	1	5	7	2	5	6	39

- Tool(s) offering best support: CASE-2, CASE-8
- Two tool combination(s) offering best support: N/A
- Requirements not supported by any tool: None

Table A.21: Tool Features in Supporting Completeness of CODE

Tools	Input to database	Cross-Referencing
CASE-1	✓	
CASE-2	✓	
CASE-3	✓	✓
CASE-4		
CASE-5	✓	✓
CASE-6		✓
CASE-7	✓	
CASE-8		✓

- Tool(s) offering best support: CASE-3, CASE-5
- Two tool combination(s) offering best support: N/A

Table A.22: Tool Features in Supporting Correctness of CODE

Tools	Req. to code	Req. to test	Test description
CASE-1	✓	✓	
CASE-2	✓	✓	✓
CASE-3	✓		
CASE-4			
CASE-5			
CASE-6	✓		✓
CASE-7	✓	✓	✓
CASE-8	✓	✓	✓

- Tool(s) offering best support: CASE-2, CASE-8, CASE-7
- Two tool combination(s) offering best support: N/A

Table A.23: Tool Features in Supporting Predictability and Robustness of CODE

Tools	Scenario-Based Analysis	Test Descriptions and Procedures
CASE-1		
CASE-2	✓	✓
CASE-3		
CASE-4		
CASE-5		
CASE-6		✓
CASE-7		✓
CASE-8	✓	✓

- Tool(s) offering best support: CASE-2, CASE-8
- Two tool combination(s) offering best support: N/A

Table A.24: Tool Features in Supporting Consistency of CODE

Tools	Consistency Checking
CASE-1	✓
CASE-2	✓
CASE-3	✓
CASE-4	✓
CASE-5	✓
CASE-6	✓
CASE-7	✓
CASE-8	✓

- Tool(s) offering best support: All
- Two tool combination(s) offering best support: N/A

Table A.25: Tool Features in Supporting Structuredness of Code

Tools	Structured Analysis
CASE-1	
CASE-2	✓
CASE-3	✓
CASE-4	✓
CASE-5	✓
CASE-6	
CASE-7	✓
CASE-8	✓

- Tool(s) offering best support: All except CASE-1, CASE-6
- Two tool combination(s) offering best support: N/A

Table A.26: Tool Features in Supporting Verifiability of CODE

Tools	*Hardware/Software Context*	Software Process Control
CASE-1	✓	✓
CASE-2	✓	✓
CASE-3	✓	✓
CASE-4		✓
CASE-5	✓	✓
CASE-6		✓
CASE-7		✓
CASE-8	✓	✓

- Tool(s) offering best support: All except CASE-6, CASE-4, CASE-7
- Two tool combination(s) offering best support: N/A

Table A.27: Tool Features in Supporting Modifiability of CODE

Tools	Revision (E.8)	Reconstruction (E.9)	Difference (E.7)	History (E.6)	Database (E.2)	Access (E.3)	Configuration (E.10) (E.5)	Referencing
CASE-1					✓	✓		
CASE-2	✓	✓	✓	✓	✓	✓		
CASE-3					✓	✓		✓
CASE-4	✓						✓	
CASE-5					✓	✓		✓
CASE-6	✓		✓	✓	✓	✓		✓
CASE-7	✓		✓	✓	✓	✓	✓	
CASE-8	✓		✓	✓	✓	✓	✓	✓

- Tool(s) offering best support: CASE-8
- Two tool combination(s) offering best support: CASE-2/CASE-8

Table A.28: Tool Features in Supporting Traceability of Code

Tools	Req. to Code	History	Cross Referencing
CASE-1	✓		
CASE-2	✓	✓	
CASE-3	✓		✓
CASE-4			
CASE-5			✓
CASE-6	✓	✓	✓
CASE-7	✓	✓	
CASE-8	✓		✓

- Tool(s) offering best support: CASE-6
- Two tool combination(s) offering best support: N/A

Table A.29: Tool Features in Supporting Understandability of Code

Tools	Auto	Support Documentation	Cross
CASE-1	✓	✓	
CASE-2			
CASE-3	✓	✓	✓
CASE-4	✓	✓	
CASE-5	✓	✓	✓
CASE-6	✓	✓	✓
CASE-7		✓	
CASE-8	✓	✓	✓

- Tool(s) offering best support: CASE-3, CASE-5, CASE-6, CASE-8
- Two tool combination(s) offering best support: N/A

Table A.30: Tools Support Code

Tool	completeness	correctness	predictability, robustness	consistency	structuredness	verifiability	modifiability	traceability	understandability	Total
CASE-1	1	2	0	1	0	2	2	1	2	11
CASE-2	1	3	2	1	1	2	6	2	0	18
CASE-3	2	1	0	1	1	2	3	2	3	15
CASE-4	0	0	0	1	1	1	2	0	2	7
CASE-5	2	0	0	1	1	2	3	1	3	13
CASE-6	1	2	1	1	0	1	6	3	3	18
CASE-7	1	3	1	1	1	1	6	2	1	17
CASE-8	1	3	2	1	1	2	7	2	3	22

- Tool(s) offering best support: CASE-8
- Two tool combination(s) offering best support: N/A
- Requirements not supported by any tool: None

A.2 Checklists of CASE Tools for Testing

A.2.1 Tools Supporting Complexity Measurement

Complexity measurement analyzers calculate the complexity of software elements (e.g. subprogram, module, program) from some associated characteristics. These tools provide guidance toward potential problem areas in the code (those areas with high complexity). Some of these tools provide test path information for developing tests. Tools which calculate code statistics, McCabe's cyclomatic complexity metric, Halstead's Software Science metrics, and complexity metrics based on a summation of code element complexities are of particular interest. The definitions of the metrics are as following:

- Code metrics usually measure the number of lines, number of source line of code, number of comments, etc.
- McCabe's cyclomatic complexity metric [51] measures the complexity of control structure of a program.
- Halstead's Software Science metrics [52] measure the complexity of a program based on the number of unique and total operators and operands.

Complexity metrics could be indicators of errors in the code. For example, in Halstead's Software Science metrics, predicted bugs is a direct function of the size of an implementation, which can be thought of as the number of bits necessary to express it. Our recent research demonstrated that the extended Halstead's metrics (we tailored it to avionics software [53]) correlate with fault density in real-time programs.

Table A.31 summarizes the available tools supporting complexity measurement and their capabilities, where "A", "C" and "F" stand for programming languages Ada, C and Fortran respectively.

Table A.31: Tools Supporting Complexity Measurement

Tool	Language	Code	McCabe	Halstead	Others
CASE-9	A/C/F	✓	✓		
CASE-10	Ada	✓	✓		✓
CASE-11	Ada	✓			
CASE-12	A/C/F	✓	✓	✓	✓
CASE-13	A/C/C++/F	✓	✓	✓	
CASE-14	A/C/F	✓			

A.2.2 Tools Supporting Syntax and Semantics Analysis

Tools supporting syntax and semantics analysis operate on the source code without regard to the executability of the program. These tools flush out as many errors as possible before testing the executable code. They can be used to identify coding errors and noncompliance (compilers do not generally provide sufficient analysis of the code to ensure consistency, portability, propose code usage, completeness, etc.). Their capabilities consist of the following:

- Cross references provide referencing of entities to other entities.
- Structural analysis provides call trees and detect structural flaws within a program (e.g., circular calls and unreachable code).
- Variable analysis checks initialization, usage, types, and scope of variables.
- Interface diagnostics checks consistency of formal and actual parameters of subprogram calls, and input, output and system interfaces.

Table A.32 summarizes the available tools supporting syntax and semantics analysis and their capabilities, where "A", "C" and "F" stand for programming languages Ada, C and Fortran, respectively.

Table A.32: Tools Supporting Syntax and Semantics Analysis

Tool	Language	Cross Reference	Structural Analysis	Variable Analysis	Interface Diagnostics
CASE-9	A/C/F		✓		
CASE-10	Ada		✓		
CASE-11	Ada	✓	✓		
CASE-12	A/C/F		✓	✓	
CASE-14	A/C/F	✓	✓		

A.2.3 Tools Supporting Test Coverage Analysis

Test coverage analysis tools assess test adequacy measures associated with the program structural elements. Coverage analysis is useful in structural testing which attempts to execute each statement, branch, path, or module.

Table A.33 summarizes the available tools supporting coverage analysis.

Table A.33: Tools Supporting Test Coverage Analysis

Tool	Language	Statement	Branch	Path	Module
CASE-11	Ada	✓	✓		✓
CASE-12	A/C/F	✓	✓		
CASE-15	A/C/F	✓	✓		✓
CASE-16	A/C/C++/F		✓	✓	✓
CASE-17	A/C/F	✓	✓		✓

A.2.4 Tools Supporting Regression Test

Regression testing verifies that only desired changes are present in modified programs. Ideally, all test cases should be re-executed and the results re-evaluated from unit-level through system-level testing with each required change. However, schedule and cost constraints almost always prevent this from occurring when modifying large software systems.

Tools have been developed which provide automated facilities for performing extensive regression testing. These tools execute various test cases using pre-recorded keystroke inputs and then compare actual results of the current test session with expected results.

Regression analysis tools are characterized by the high-level capabilities of capture, replay, and compare:

- *Capture* is the capability of recording the inputs (scripts) and outputs (benchmarks) of a test session. Typically, inputs consist of keyboard inputs, and outputs consist of terminal screen displays. Text editors create and modify script files or test drivers for some regression tools.

- *Replay* is the capability of reissuing pre-recorded inputs (playing back the script). This ensures that test case inputs are the same as in previous tests and minimizes the tedious, error-prone procedures that must be executed.

- *Compare* is the capability of determining that the actual results of the current test session are the same as those of a previous test session (benchmark). This allows the tester to focus his attention on resolving discrepancies instead of locating the discrepancies. Some regression tools can only compare text outputs, while others can compare graphics output.

Table A.34: Tools Supporting Regression Testing

Tool	Platform	Capture			Replay	Compare
		key	mouse	screen		
CASE-18	DOS	✓		Text	Script	Text
CASE-19	DOS	✓		Text	Script	Text
CASE-20	DOS	✓		Text		Text
CASE-21	DOS	✓		Text	Script	Text
CASE-22	UNIX	✓	✓	Graphics	Script	bit-mapped
CASE-23	VMS	✓	✓	Graphics		bit-mapped

A.2.5 Tools Supporting Test Data Generation

Test data generation is the process of identifying a set of test data which satisfies given testing criterion. It is one of the most difficult and time consuming parts of software testing. Unfortunately automatic test data generation is still in the research stage. Table A.35 summarizes some of the tools, commercially available or research prototypes, for test data generation. These tools generate test data using different test selection criteria. All these tools have limited

capabilities and can only be used to generate some of the test cases of the complete test set. Among the tools listed in Table A.35 table CASE-25 [54] is research prototypes, the others is commercially available. The "*" in the column "Language" stands for "language independent."

Table A.35: Tools Supporting Test Data Generation

Tool	Language	System	Test Selection Criterion
CASE-24	*	Unix/DOS/VMS	functional & random testing
CASE-25	C/Fortran	Unix	mutation testing

Table A.36: Codes for CASE Tools

Code	Tool Name
CASE-1	Aisle/Cisle
CASE-2	ObjectMaker
CASE-3	PowerTools
CASE-4	ProMod
CASE-5	Software through Pictures
CASE-6	Super CASE
CASE-7	TAGS Case 2
CASE-8	Teamwork
CASE-9	ACT/BAT
CASE-10	AdaMAT
CASE-11	ATVS
CASE-12	Logiscope
CASE-13	PC_Metric
CASE-14	VAX SCA
CASE-15	SAW
CASE-16	STW/COV
CASE-17	VAX PCA
CASE-18	Automator
CASE-19	AutoTester
CASE-20	Bloodhound
CASE-21	Check*Mate
CASE-22	STW/REG
CASE-23	DEC/DTM
CASE-24	T
CASE-25	Mothra

Appendix B

Examples of Fault-Tolerant Safety System Design

B.1 Background

The body of this report points out that the highest degree of reliability for fault tolerant systems is at present achieved with functional diversity. On the other hand, an essential advantage of digital over analog processing is the ability to perform a comprehensive evaluation of the state of a plant by taking into account the indications from a number of diverse sensors. In order to achieve this advantage some of the independence of the diverse indications is sacrificed in the common evaluation algorithm. This section

- presents a methodology for the comparative evaluation of fault tolerant hardware/software architectures that incorporate functional diversity
- outlines examples of three architectures that maintain the multi-channel approach found in current analog plant protection systems and therefore minimize the difficulties of transitioning from analog to digital technology (the term outline is deliberately used to indicate that only essentials are shown and evaluated)
- identifies hardware/software fault tolerance features that are evaluated as particularly beneficial in this context.

Most current systems employ analog technology in the form shown in Figure B.1 in which operation of the protective system is controlled by individual plant variables processed in hardware redundant channels. Direct translation of this approach into digital equipment is possible (e.g., Ontario Hydro is developing a digital trip meter [55]), but this does not achieve

the advantages of digital technology in hardware integration (important for reduction of maintenance costs) and in the comprehensive evaluation of the plant state that was mentioned above.

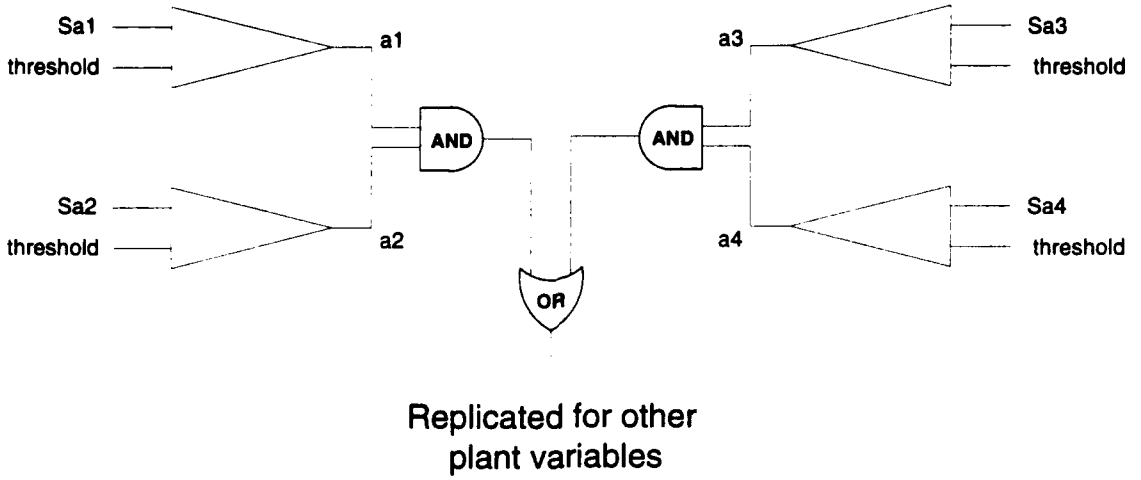


Figure B.1: Analog Implementation

The architecture of Scheme I, shown in Figure B.2, is specifically intended to achieve these advantages: three different sensors are processed in each computer, and the operation of the protective system is controlled by a single output from each computer that represents an assessment of the plant state based on the combined information from the three sensors (the structure of the computer program is discussed later). An alternative approach, Scheme II, is shown in Figure B.3. Here three signals are processed in each computer but the outputs are combined externally with AND gates that activate the protective function when at least two out of four outputs derived from a given sensor type call for trip. This achieves hardware integration but sacrifices the advantage of comprehensive evaluation. The notation used in the figures is explained below.

Sai input data from sensor *i* for plant variable *a* (e.g. temperature).

Sbi input data from sensor *i* for plant variable *b* (e.g. pressure).

Sci input data from sensor *i* for plant variable *c* (e.g. flux).

ai output of Trip function using data from sensor *i* of type *a*.

bi output of Trip function using data from sensor *i* of type *b*.

ci output of Trip function using data from sensor *i* of type *c*.

xj output of combined Trip function from computer *j*.

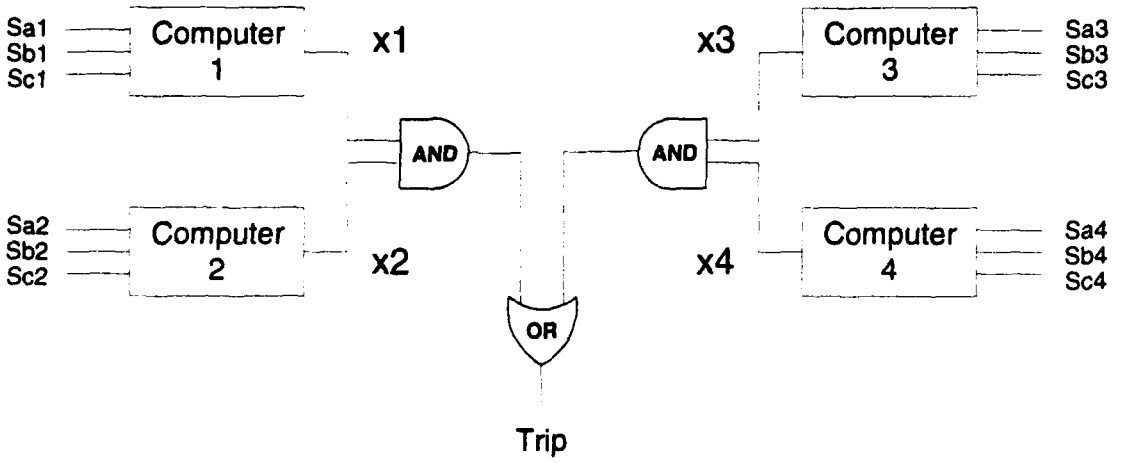


Figure B.2: Scheme I System for Generation of Trip Signal

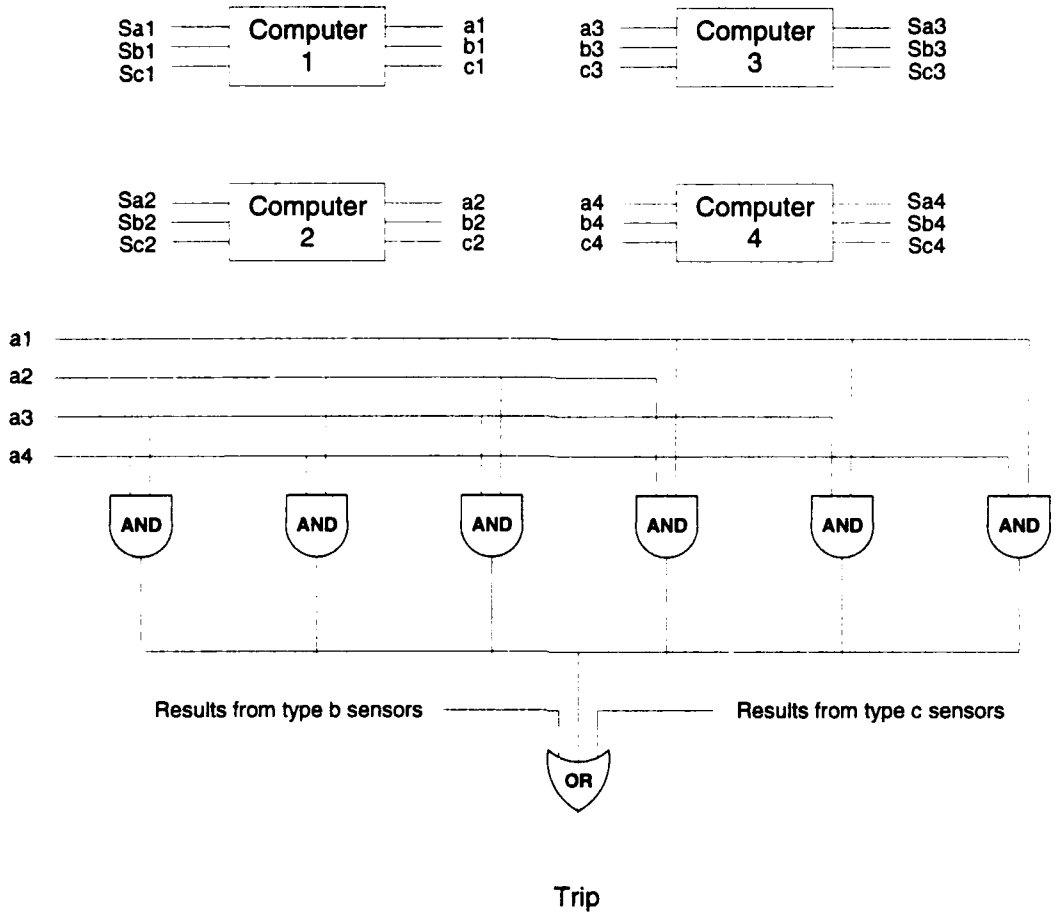


Figure B.3: Scheme II System for Generation of Trip Signal

B.2 Definitions and Assumptions

1. *System failure:*
 - Type-1: System fails to generate Trip signal given that Trip is necessary.
 - Type-2: System generates false Trip signal when Trip is unnecessary.
2. *Sensor Error:* Sensor supplies erroneous data leading to a wrong decision of the Trip function.
3. *Trip Function Error:* Software generates erroneous computational result leading to a wrong decision on Trip action. The assumed software structure for Scheme I is shown

in Figure B.4. The common State Evaluation function is assumed to contain between 1% and 10% of the failure probability for an individual sensor chain. Scheme II does not include a common State Evaluation function.

4. *Computer Failure:* Computer becomes non-operational and is perceived as "stuck on zero."

5. *System Parameters:*

 - Redundant computers have the same probabilities of becoming non-operational.
 - Redundant sensors have the same error probabilities.
 - Redundant Trip functions have the same error probabilities.

6. *Single Gate Error:* Negligible likelihood (can be achieved by self-checking gates).

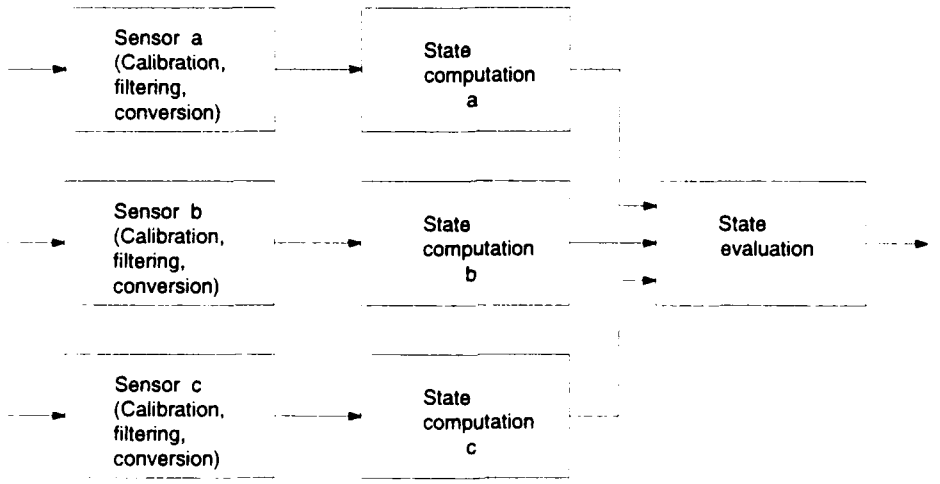


Figure B.4: Software Structure for Scheme I

B.3 The Models

B.3.1 Notations

The following notations are used in the mathematical models for computing system unreliability.

q_s = P(sensor error).
q_f = P(Trip function error).

q_c = P(computer failure).
q_d = P(State Evaluation failure).
q_{dc} = P(Common mode failure of state functions).

Note that q_d and q_{dc} are for Scheme I only.

Our quantitative evaluations shown in the following sections are based on the parameter assignment shown in Table B.1. (Some of the parameters become variables in certain evaluations.) The assignment is made on the basis of findings in equivalent digital systems including space shuttle avionics systems.

Table B.1: Assignments of Parameters for Quantitative Evaluations

Parameter	Value
q_s	0.00001
q_f	0.00001
q_c	0.00001
q_d	0.000001
q_{dc}	0.0000001

B.3.2 Failure Type-1

Scheme I We define the following indices.

n: index to sensor type, $n \in \{1 \ldots 3\}$.
j: index to channel, $j \in \{1 \ldots 4\}$.
k: index to pair, $k \in \{1 \ldots 2\}$.

We have

P(Type-1 Failure)

$$= \prod_{k=1}^{2} P(pair\ k\ fails)$$

$$= \left(\prod_{j=1}^{2} P(channel\ j\ fails) + q_{dc}(1-q_c)^2 \right) \left(\prod_{j=3}^{4} P(channel\ j\ fails) + q_{dc}(1-q_c)^2 \right)$$

And

$$P(channel\ j\ fails) = \left\{\left(\prod_{n=1}^{3}(q_{sn}+q_{fn})\right)(1-q_d-q_{dc})+q_d\right\}\cdot(1-q_c)+q_c.$$

Therefore,

$$P(Type\text{-}1\ failure) = \prod_{k=1}^{2}\left\{2\cdot\left\{\left[(q_s+q_f)^3\cdot(1-q_d-q_{dc})+q_d\right](1-q_c)+q_c\right\}+q_{dc}(1-q_c)^2\right\} \quad (B.1)$$

Scheme II

P(Type-1 failure) = P(at least 3 computers fail) +
P(at most 2 computers fail & system fails to generate "Trip").

$$P(at\ least\ 3\ computers\ fail) = q_c^4+\binom{4}{3}q_c^3(1-q_c).$$

P(at most 2 computers fail & system fails to generate "Trip") =

$$\begin{aligned}&(1-q_c)^4\left[(q_s+q_f)^4+\binom{4}{3}\cdot(q_s+q_f)^3\cdot(1-q_s-q_f)\right]^3+\\&\binom{4}{3}(1-q_c)^3\cdot q_c\left[(q_s+q_f)^3+\binom{3}{2}\cdot(q_s+q_f)^2\cdot(1-q_s-q_f)\right]^3+\\&\binom{4}{2}(1-q_c)^2\cdot q_c^2\left[(q_s+q_f)^2+\binom{2}{1}\cdot(q_s+q_f)\cdot(1-q_s-q_f)\right]^3\end{aligned}$$

Therefore

$$\begin{aligned}P(\text{Type-1 failure}) = {} & 4\cdot q_c^3-3\cdot q_c^4+\\&(1-q_c)^4\left[(q_s+q_f)^4+4\cdot(q_s+q_f)^3\cdot(1-q_s-q_f)\right]^3+\\&4\cdot(1-q_c)^3\cdot q_c\left[(q_s+q_f)^3+3\cdot(q_s+q_f)^2\cdot(1-q_s-q_f)\right]^3+\\&6\cdot(1-q_c)^2\cdot q_c^2\left[(q_s+q_f)^2+2\cdot(q_s+q_f)\cdot(1-q_s-q_f)\right]^3\end{aligned} \quad (B.2)$$

B.3.3 Failure Type-2

Scheme I Again we use the notations:

n: index to sensor type, $n \in \{1 \ldots 3\}$.
j: index to channel, $j \in \{1 \ldots 4\}$.
k: index to pair, $k \in \{1 \ldots 2\}$.

$$\text{P(Type-2 Failure)} = \sum_{k=1}^{2} P(\textit{pair } k \textit{ fails})$$
$$= \left\{ \prod_{j=1}^{2} P(\textit{channel } j \textit{ fails}) + q_{dc}(1-q_c)^2 \right\} +$$
$$\left\{ \prod_{j=3}^{4} P(\textit{channel } j \textit{ fails}) + q_{dc}(1-q_c)^2 \right\}$$

And

$$P(\textit{channel } j \textit{ fails}) = \left\{ \left[\sum_{n=1}^{3} (q_{sn} + q_{fn}) \right] \cdot (1-q_d) + q_d \right\} \cdot (1-q_c).$$

Therefore,

$$\text{P(Type-2 Failure)} = \sum_{k=1}^{2} \left\{ \left\{ [3 \cdot (q_{sn} + q_{fn}) \cdot (1-q_d) + q_d] \cdot (1-q_c) \right\}^2 + q_{dc}(1-q_c)^2 \right\} \qquad \text{(B.3)}$$
$$= 2 \cdot \left\{ \left\{ [3 \cdot (q_{sn} + q_{fn}) \cdot (1-q_d) + q_d] \cdot (1-q_c) \right\}^2 + q_{dc}(1-q_c)^2 \right\}.$$

Scheme II

P(Type-2 failure) = P(at least 2 computers up & system generates false Trip signal)
= P(4 computers up & system generates false Trip signal) +
P(3 computers up & system generates false Trip signal) +
P(2 computers up & system generates false Trip signal).

P(4 computers up & system generates false Trip signal) =

$$(1-q_c)^4 \cdot 3 \cdot [(q_s + q_f)^4 + \binom{4}{3} \cdot (q_s + q_f)^3 \cdot (1 - q_s - q_f) +$$
$$\binom{4}{2} (q_f + q_s)^2 \cdot (1 - q_s - q_f)^2],$$

P(3 computers up & system generates false Trip signal) =

$$\binom{4}{3}(1-q_c)^3 \cdot q_c \cdot 3 \cdot \left[(q_s+q_f)^3 + \binom{3}{2}(q_s+q_f)^2 \cdot (1-q_s-q_f)\right]$$

P(2 computers up & system generates false Trip signal) = $\binom{4}{2}(1-q_c)^2 \cdot q_c^2 \cdot 3 \cdot (q_s+q_f)^2$

Therefore,

$$\begin{aligned} P(\text{Type-2 failure}) = {} & (1-q_c)^4 \cdot 3 \cdot [(q_s+q_f)^4 + \\ & 4 \cdot (q_s+q_f)^3 \cdot (1-q_s-q_f) + 6 \cdot (q_f+q_s)^2 \cdot (1-q_s-q_f)^2] + \\ & 4 \cdot (1-q_c)^3 \cdot q_c \cdot 3 \cdot [(q_s+q_f)^3 + 3 \cdot (q_s+q_f)^2 \cdot (1-q_s-q_f)] + \\ & 6 \cdot (1-q_c)^2 \cdot q_c^2 \cdot 3 \cdot (q_s+q_f)^2. \end{aligned} \qquad \text{(B.4)}$$

B.4 Comparisons of Schemes I and II

From Eq.'s (B.1) and (B.2), we notice that computer unreliability (q_c) is the major factor contributing to type-1 system failure. The reason is that

1. Compared with sensors, computers have a lower degree of redundancy. That is, for sensors there are two levels of redundancy while computers have only a single level. Therefore, computers become the reliability "bottle-neck," and systems are more sensitive to computer failures.

2. The analysis assumes that all computer failures are *stuck-at-zero* and multiple ones can result in type-1 system failures. While this is pessimistic there are no creditable statistics that validate a significantly different assumption. By contrast, even multiple sensor failures are prevented from resulting in type-1 system failures by the state evaluation function in Scheme I and by the multiple OR gates in Scheme II.

Accordingly, the probability of type-1 failures is primarily evaluated as a function of computer failure probability as shown in Figure B.5. For type-2 failures the most important cause is a common software failure in the state computation function for a given sensor type (see Figure B.4; the left and middle function blocks in this figure are also present in Scheme II). The probability of type-2 failure is shown in Figure B.6.

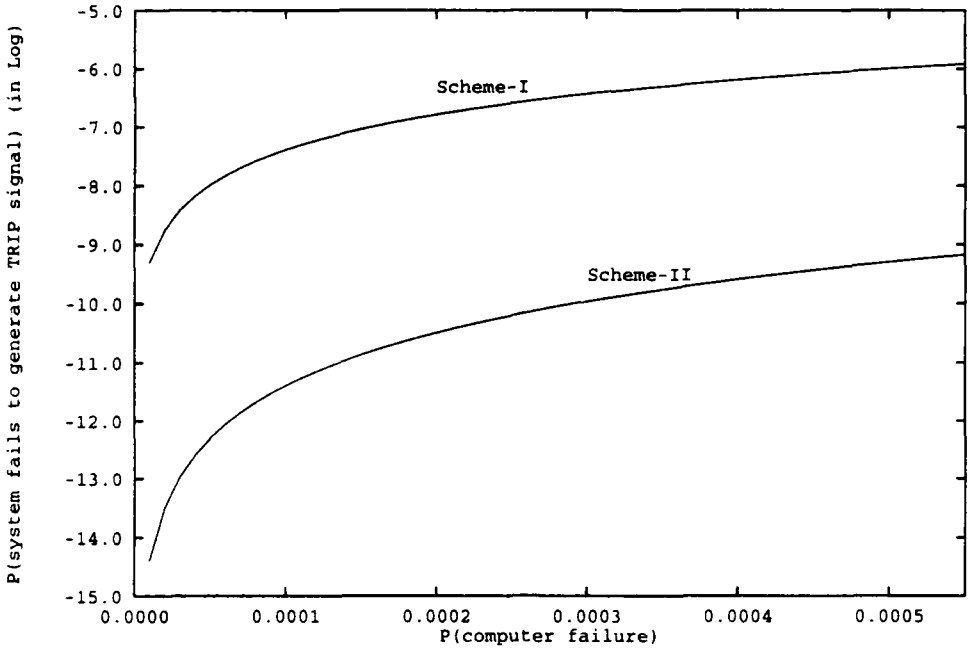


Figure B.5: Probability of Type-1 Failure as a Function of Hardware Failure

Figure B.5 shows that Scheme II is superior to Scheme I. By looking into Eq.'s (B.1) and (B.2), it is observed that the dominant term of the former is $4 \cdot q_c^2$ while that of the latter is $4 \cdot q_c^3$. This is due to the following reasons. In Scheme I, any two computer failures across pairs lead to Type-1 failures. On the other hand, in Scheme II, failure of any two computers is not a sufficient condition for Type-1 failure, but three or more is. The evaluation here is based on low probability of independent and common mode failures of the state function. When those probabilities become high, the Type-1 failure probability difference between Schemes I and II will be even more significant. Although the two schemes primarily have the same amount of redundancies, Scheme II exploits the redundant resources in a more effective manner so achieves reliability goal better.

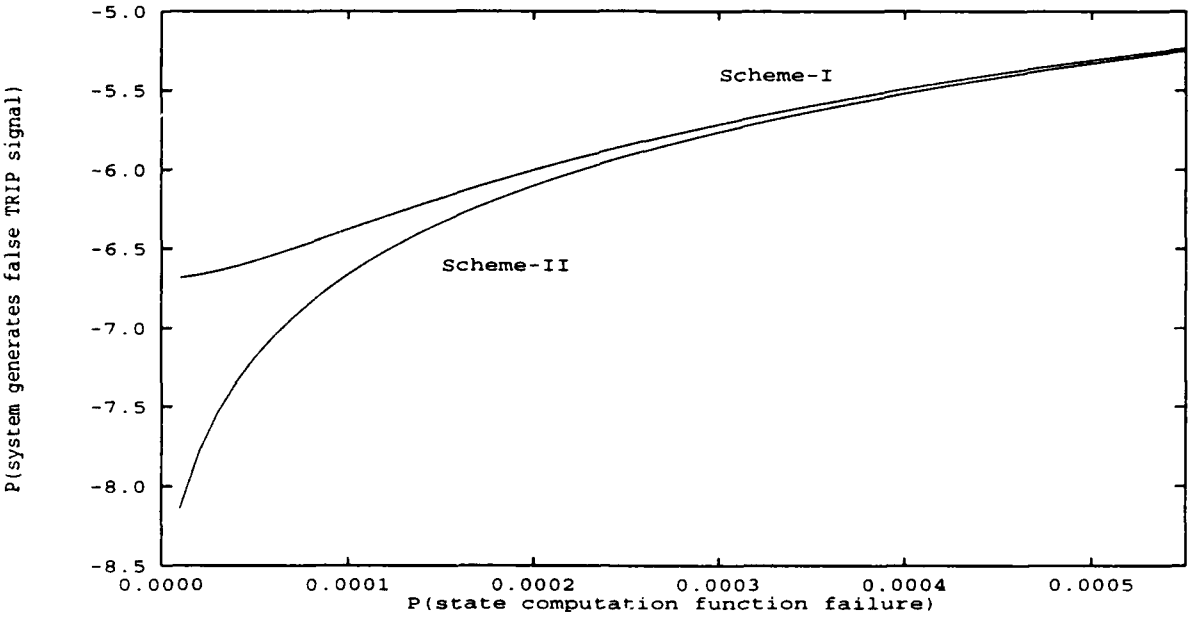


Figure B.6: Probability of Type-2 Failure as a Function of Software Failure

Type-2 failure is rather insensitive to computer failure probability (primarily due to our assumption on the computer failure behavior, i.e., "stuck at zero"). Therefore, we evaluate the probabilities of type-2 failures as a function of software (Trip function) failure. Figure B.6 shows that the type-2 failure probabilities of the two schemes are getting close when the Trip function failure probability is high. By looking into Eq.'s (B.3) and (B.4), it is observed that the dominant term of the former is $2q_{dc}(1-q_c)^2$ when Trip function failure probability is low, and the dominant term becomes $18(q_s+q_f)^2(1-q_d)(1-q_c)^2$ when the Trip function failure probability is high while the dominant term of the latter is always $18(q_s+q_f)^2(1-q_s-q_f)^2(1-q_c)^4$, which mathematically explain the phenomenon.

It is seen that Scheme II offers greater fault tolerance with respect to both type-1 and type-2 failures, and that this advantage is particularly pronounced for the more critical type-1 failures. However, Scheme II does not provide a comprehensive state evaluation and thus sacrifices a potential benefit of the digital implementation. The following section describes a modification of Scheme I which retains its functional advantage while offering appreciably higher reliability.

B.5 Scheme III - Extended Distributed Recovery Block (EDRB)

The top level architecture of Scheme III is shown in Figure B.7, and the structural similarity to Scheme I (Figure B.2) is easily recognized. However, there are also a number of significant differences: the AND gates at the outputs of the individual computer pairs are replaced by XOR gates (that actually represent a switching function described below), and the top and bottom computers in each pair initially run different software versions, called X and Y. The Extended Distributed Recovery Blocks (EDRB) has been developed under Department of Energy sponsorship to permit digital techniques to be applied to the control of critical processes [56]. The underlying fault tolerance techniques are based on extensions of the Distributed Recovery Block [57]. The EDRB has been installed and integrated with a nuclear process control application at the Argonne West Experimental Breeder Reactor II.

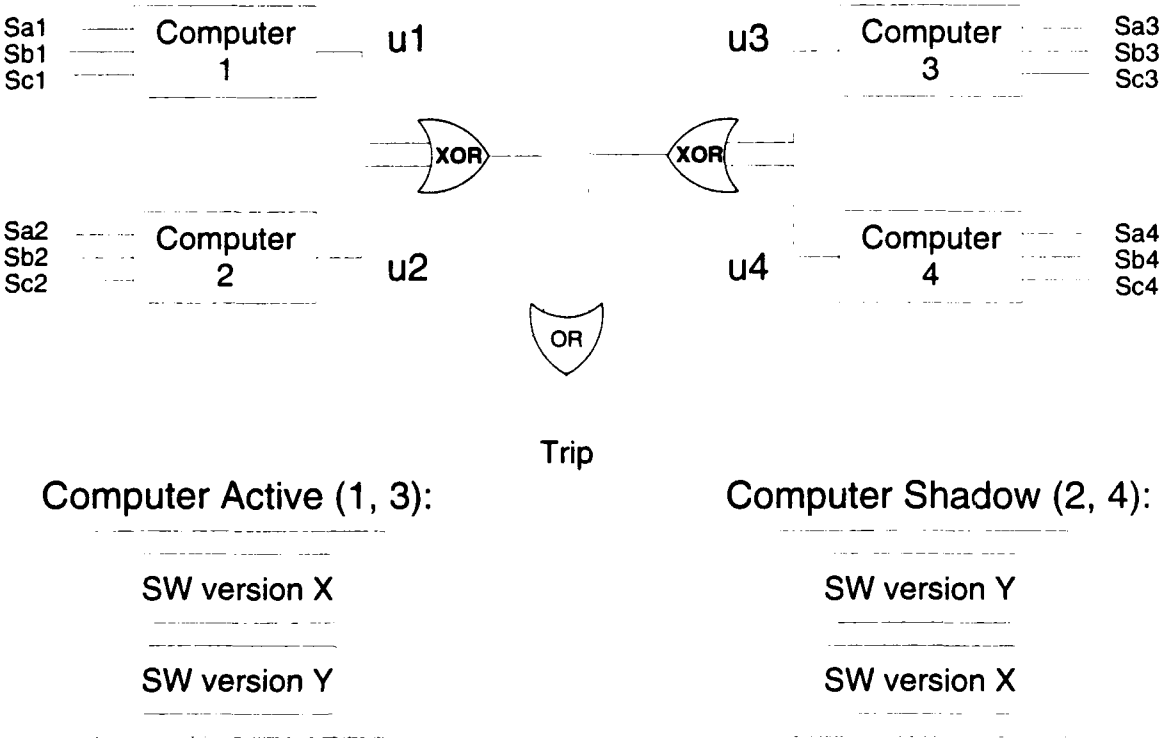


Figure B.7: Overview of the EDRB

A distinguishing feature of the recovery block approach to software fault tolerance is the acceptance test (AT), a rigorously designed set of assertions that must be satisfied before results of a computation are communicated outside the computer. In many cases the problem statement or the specification of the software impose conditions that must be met at the completion of program execution. These conditions may be used to construct the acceptance test. *Testing for*

satisfaction of requirements is usually most effective when carried out on small segments of code. *Accounting checks* are suitable for transaction-oriented applications with simple mathematical operations such as airline reservation systems, library records, and the control of hazardous materials. The simplest form of accounting checks is the checksum. *Reasonable tests* detect software failures by use of pre-computed ranges, expected sequences of program states, or other relationships that are expected to prevail. Reasonable tests are suitable in control or switching systems where physical constraints can determine the range of possible outcomes. The presence of this test greatly reduces the probability that a single software failure might lead to a system level type-2 (unnecessary trip) failure, and this permits the AND gates in Figure B.2 to be replaced with XOR gates in Figure B.7. With this modification the probability of the architecture to respond correctly to necessary trips is increased, and, conversely, the probability of a type-1 failure is significantly increased. To understand the quantitative evaluation of these capabilities it will be helpful to explain the operation of the EDRB in more detail.

A high level version of EDRB operation is shown in Figure B.8. Normally version X of the software runs in computer 1, and its output is subjected to the acceptance test (AT). If the test is passed (T exit in the figure), the result of the operation is passed to the output. If the test is not passed (F exit) the result from version Y running in computer 2 is used, provided that it has passed its acceptance test. If the acceptance test is not passed the program will be put into an exception state which is further discussed below. The transition to computer 2 will also take place when there is a hardware failure in computer 1. The immediate transition on any failure to a second computer with a different software version provides a very robust type of fault tolerance, and the ability of this architecture to handle unexpected faults will be evident below.

A more detailed view of the operation of the EDRB is shown in Figure B.9. On failure of the version X acceptance test in computer 1 the transition to computer 2 takes the form of an enabling signal going through the OR gate in computer 1 to the AND gate in computer 2 which then permits the results of version Y to be fed to the output. On detection of a hardware failure in computer 1 the same enabling signal is transmitted via the OR gate to computer 2.

If version Y in computer 2 does not pass the acceptance test the exception handling indicated in Figure B.9 will be entered. It consists initially of running version X in computer 2, and if this is successful permitting its result to be furnished to the output. If that acceptance test also fails version Y can be run in computer 1 and its result furnished to the output. Finally, if all versions fail an alarm condition is created which may cause a reactor trip or permit computers 3 and 4 to run in a stand-alone mode, depending on the specific operational requirements.

The quantitative analysis of Scheme III will now be explained with the aid of fault trees shown in Figures B.10 and B.11. The triangles in Figure B.10 indicate continuations in Figure B.11.

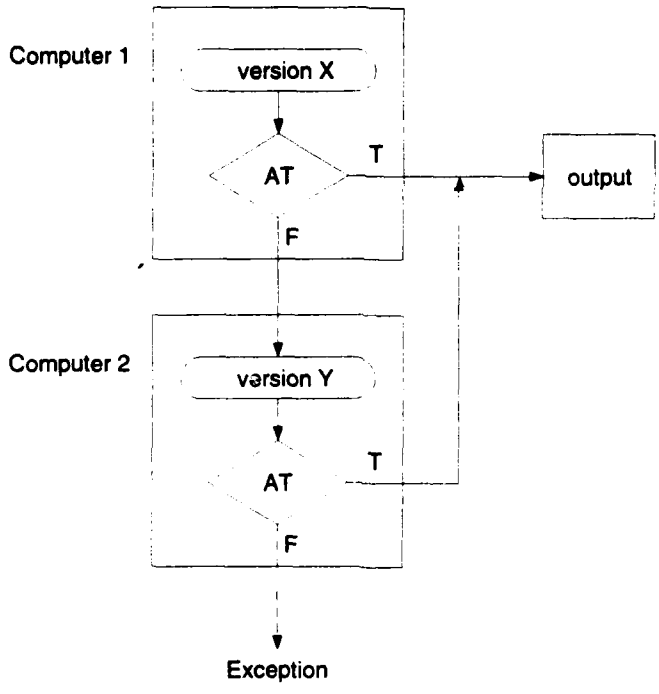


Figure B.8: EDRB Operation (high level)

Type-1 Failure The following assumptions have been made: when all three sensor chains in a channel fail to indicate a necessary trip, it is unlikely that the state function or the acceptance test will alert a Trip.

$$P(Type\text{-}1\ failure)=\sum_{k=1}^{2}(pair\ k\ failure)$$

Let Q_s be the probability that one or more sensor chains indicate a necessary Trip, that is

$$Q_s=(1-q_s-q_f)^3+\binom{3}{2}(1-q_s-q_f)^2(q_s+q_f)+\binom{3}{1}(1-q_s-q_f)(q_s+q_f)^2.$$

Let the probability of related errors between the state function and the acceptance test be denoted as q_{dt}, then we can define the following notations for the case in which both nodes in a pair are up:

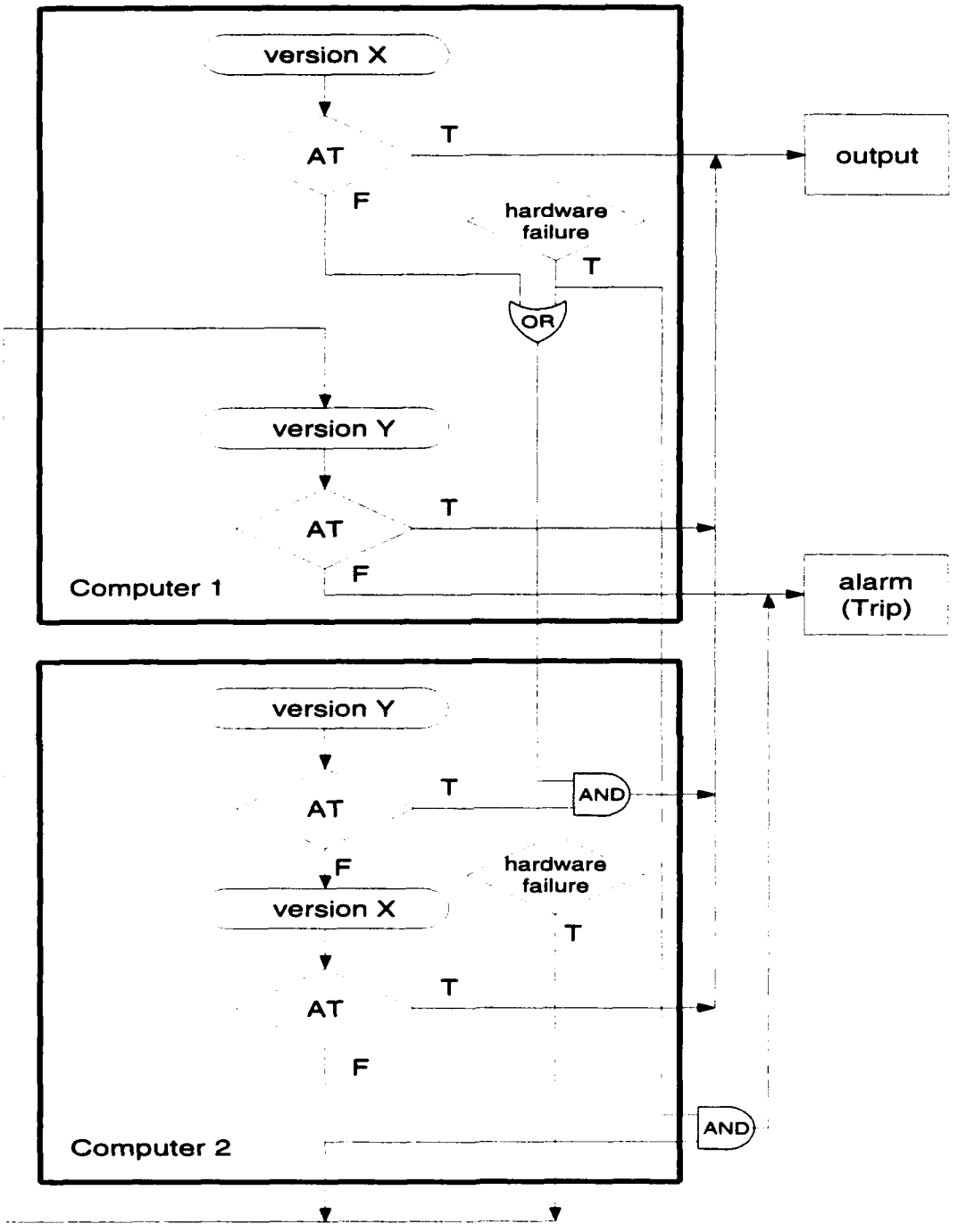


Figure B.9: EDRB Operation (low level)

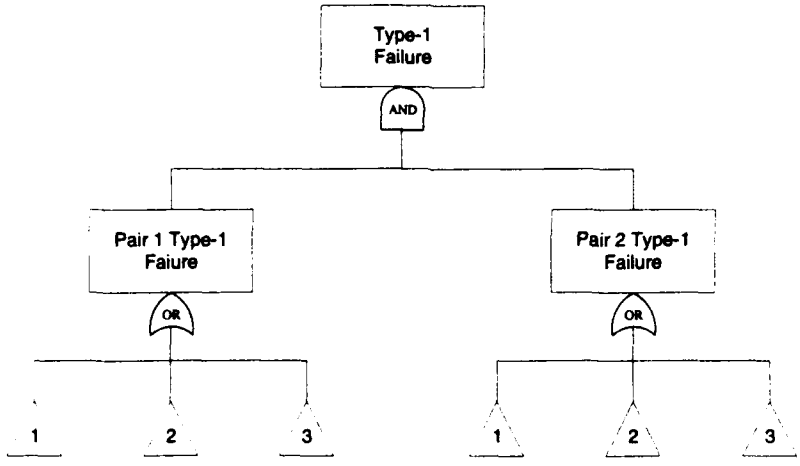


Figure B.10: Fault Tree for Type-1 Failure (I)

$$Q_{1c} = \text{P(node 1 type-1 failure)}$$
$$= (q_s + q_f)^3 + Q_s \cdot q_{dt}$$

$$Q_{1b} = \text{P(node 1 benign failure)}$$
$$= Q_s \cdot [(1 - q_d) \cdot q_t + q_d \cdot (1 - q_t)]$$

$$Q_{2c} = \text{P(node 2 software catastrophic failure)}$$
$$= (q_s + q_f)^3 + Q_s \cdot q_{dt} + Q_s \cdot [(1 - q_d) \cdot q_t + q_d \cdot (1 - q_t)] \cdot q_{dt}$$

For the case in which one node is up and the other is down we have:

$$Q_c = \text{P(up-node software catastrophic failure)}$$
$$= Q_{2c}$$

And let the probability that a computer is un-operational be denoted as Q_h. Then

$$Q_h = q_c$$

Thus we have:

$$\text{P(pair k fails)} = \text{P(failure | one node down and the other up) +}$$
$$\text{P(failure | both nodes up) + P(failure | both nodes down)}$$
$$= (Q_{1b} \cdot Q_{2c} + Q_{1c}) \cdot (1 - Q_h)^2 + \binom{2}{1} Q_c (1 - Q_h) \cdot Q_h + Q_h^2.$$

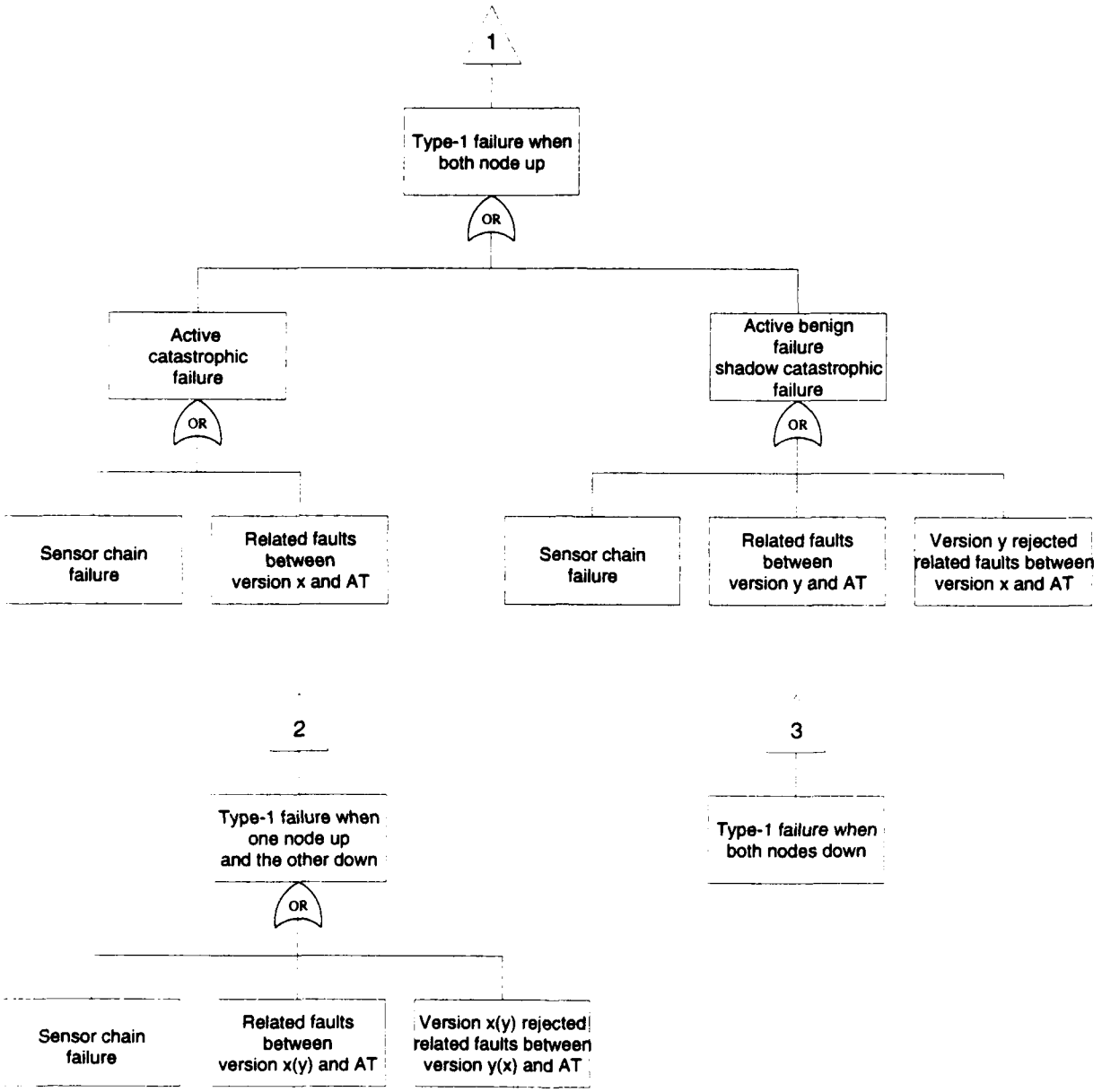


Figure B.11: Fault Tree for Type-1 Failure (II)

$$\text{P(Type-1 failure)} = \left[(Q_{1b}\cdot Q_{2c}+Q_{1c})\cdot(1-Q_h)^2+\binom{2}{1}Q_c(1-Q_h)\cdot Q_h+Q_h^{\,2}\right]^2. \qquad \text{(B.5)}$$

In the evaluation of Scheme III, the value of failure probability of the acceptance test q_t is set to 0.00005. This assignment is conservative since q_t is a critical factor for the effectiveness

of Scheme III and we relatively do not have much experience with this factor. For consistency, the probability of related failures between a program version (the state evaluation function) and the acceptance test q_{dt} is made equivalent to the probability of common mode failure q_{dc} of Scheme I.

Figure B.12 shows the comparison of the Type-1 failure probability as a function of computer failure probability. It reveals that except for extremely low computer failure probability, Scheme III is superior to the other two schemes. The reason is that Scheme III is least sensitive to multiple computer failures (while Scheme I is most sensitive). For extremely low computer failure probability, Scheme II appears to be the best because it is free of common mode or related failures which are major failure causes for Schemes I and III.

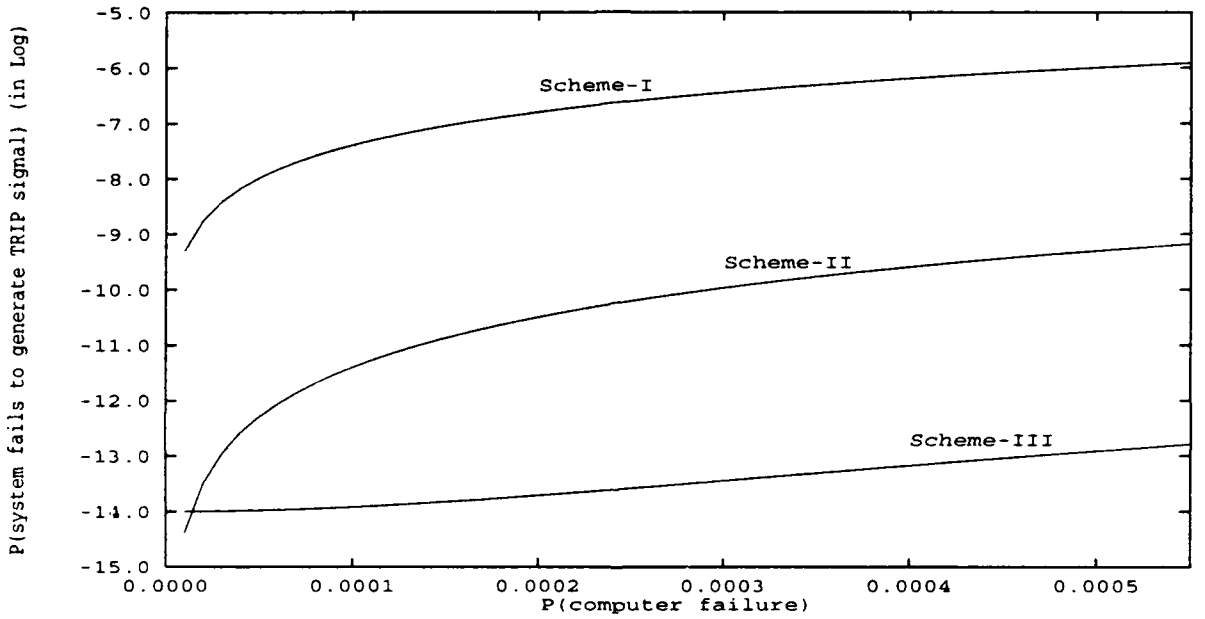


Figure B.12: Comparison of Scheme I, Scheme II and Scheme III

Figure B.13 depicts the improvement of Type-1 failure probability of the Scheme III over Scheme I, for which the failure probability is evaluated as functions of common mode failure probability q_{dc} and of related failure probability q_{dt} for the original Scheme I and Scheme III, respectively.

We observe the reduction of Type-1 failure probability due to the EDRB enhancement. The improvement is most significant when common-mode/related failure probabilities are low. The underlying reason is as follows. The dominant term for the original Scheme I (Eq. (B.1)) is $(2(q_d(1-q_c)+q_c)+q_{dc}(1-q_c)^2)^2$ while that for EDRB scheme is $((1-q_s-q_f)^3 \cdot q_{dt}(1-q_c)^2)^2$. This implies that the factors of computer failure probability and state function independent failure probability

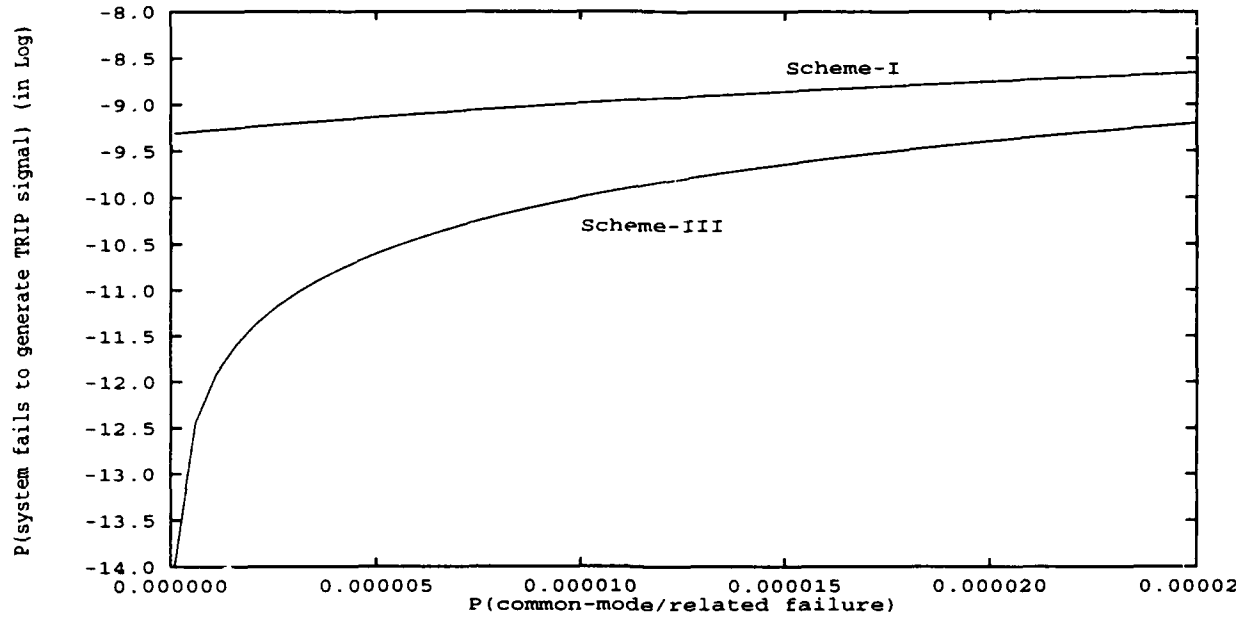


Figure B.13: Type-1 Failure Probability Improvement of Scheme III over Scheme I

affect Scheme I significantly when the common-mode failure probability is low. Those factors do not present the dominant term for the EDRB Scheme Type-1 failure probability.

Type-2 Failure The following assumption has been made: when all three sensor chains indicate an unnecessary Trip, it is unlikely that the state function or the acceptance test will detect the error.

$$P(Type\text{-}2\ failure)=\sum_{k=1}^{2}(pair\ k\ failure)$$

Let Q_s be the probability that two or fewer sensor chains indicate an unnecessary Trip, that is

$$Q_s=(1-q_s-q_f)^3+\binom{3}{2}(q_s+q_f)(1-q_s-q_f)^2+\binom{3}{1}(q_s+q_f)^2(1-q_s-q_f).$$

We define the following notations for the case in which both nodes in a pair are up:

$$\begin{aligned} Q_{12} &= \text{P(node 1 type-2 failure)} \\ &= (q_s + q_f)^3 + Q_s \cdot q_{dt} \end{aligned}$$

$$Q_{1b} = \text{P(node 1 benign failure)}$$
$$= Q_s \cdot [(1 - q_d) \cdot q_t + q_d \cdot (1 - q_t)]$$

$$Q_{22} = \text{P(node 2 type-2 failure)}$$
$$= (q_s + q_f)^3 + Q_s \cdot q_{dt} + Q_s \cdot [(1 - q_d) \cdot q_t + q_d \cdot (1 - q_t)] \cdot q_{dt}$$

For the case in which one node is up and the other is down we have:

$$Q_2 = \text{P(up-node type-2 failure)}$$
$$= Q_{22}$$

P(pair k fails) = P(failure | one node down and the other up) +
P(failure | both node up)

$$=(Q_{1b}\cdot Q_{22}+Q_{12})\cdot(1-Q_h)^2+\binom{2}{1}Q_2(1-Q_h)\cdot Q_h.$$

$$\text{P(Type-2 failure)} = 2\left[(Q_{1b}\cdot Q_{22}+Q_{12})\cdot(1-Q_h)^2+\binom{2}{1}Q_2(1-Q_h)\cdot Q_h\right] \qquad \text{(B.6)}$$

The comparison of Type-2 failure probabilities for Scheme I and Scheme III is shown in Figure B.14. The two curves virtually coincide because the dominate terms of Type-2 failure probabilities for both Scheme I and Scheme III are the first order terms of common-mode and related failure probabilities (That is, $2d_c(1-q_c)^2$ and $2(1-q_s-q_f)^3q_{dt}(1-q_c)^2$ respectively. The results imply that the improvement of Type-2 failure probability from incorporating EDRB is practically negligible.

B.6 Techniques of Architecture Selection for Fault Tolerance

We have evaluated three architectures that permit functional diversity to be introduced in a computer environment designed for easy transition from current analog plant protection systems. This final section extracts some of the methodological insights that were gained in this process.

1. Figure of Merit

 Any selection is based on a figure of merit (FOM). If no FOM is identified it must still exist in the decision maker's mind. The

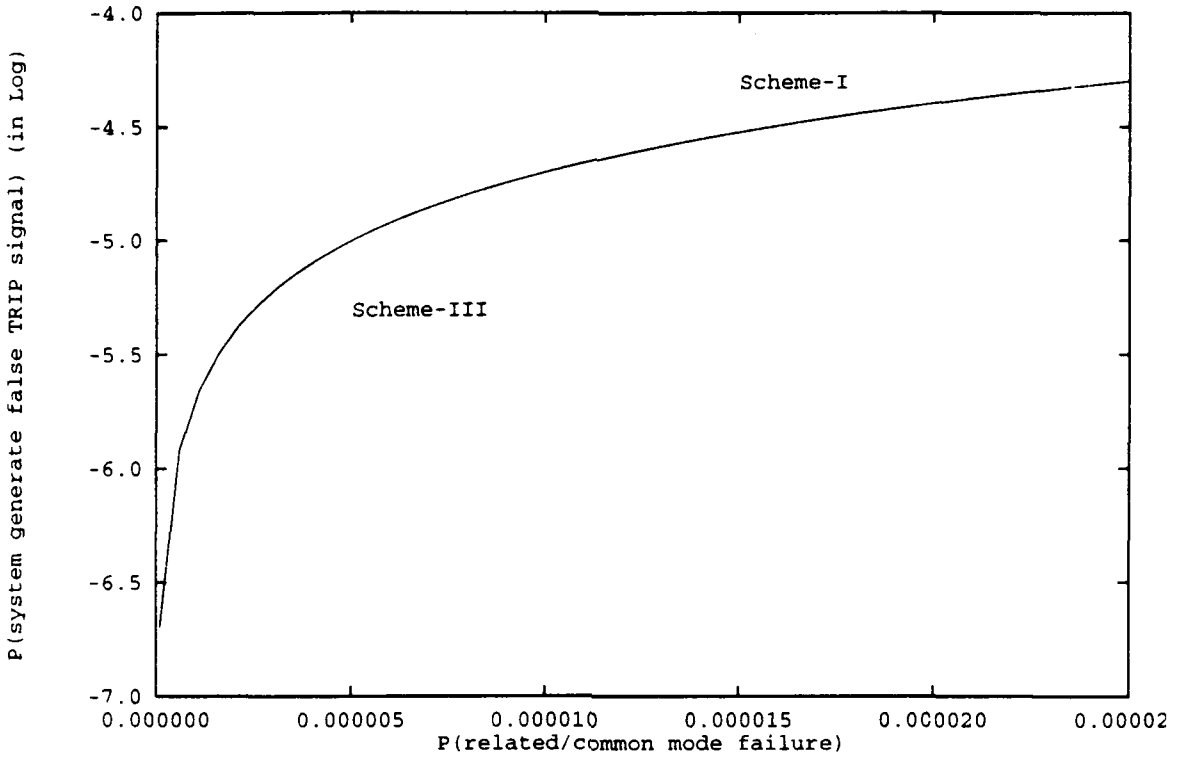


Figure B.14: Type-2 Failure Probability of Scheme I and Scheme III

implied FOM here has been minimum failure probability but taken by itself it may lead to selections that do not optimally meet the user's objectives. An example of this occurred in the comparison between Scheme I and Scheme II. The latter has significantly lower failure probability but is less suitable than Scheme I if the user needs or prefers a plant status evaluation derived from multiple sensor inputs. Other issues that may be considered in the FOM are throughput differences between architectures and the number and types of communication channels that are required for data exchange.

2. Ordering of Significant Parameters

The alternative architectures will inevitably differ in a number of parameters that enter into the FOM calculation. It is necessary to rank the parameters in their order of contribution to the FOM. In the examples worked here computer (hardware) reliability was in most cases the top ranked parameter, and thus it was used as the

independent variable in most of the comparative plots of system failure probability. Sensitivity studies can determine what values the subsidiary parameters can take before invalidating the outcome of the selection.

3. Understanding the Strengths and Weaknesses of the Selected Alternative

 The selection process should not stop with the identification of the preferred alternative. Much can be gained from examining the strengths of the preferred alternative that contributed to its selection, and of course at least as much effort should be expended on identifying and understanding its weaknesses. Thus, the selection process did not stop here when Scheme II was shown to be more reliable than Scheme I. Evaluation of what made Scheme II superior led to the exploration of other architectures, including the one described here as Scheme III which is superior to both of those evaluated earlier.

4. Software Defenses against Software Failures

 Carrying the last recommendation a step further, it will be realized that one of the strengths of Scheme III is that it has software defenses (the acceptance test) against software failures. Other architectures that embody this feature might therefore also merit consideration. The acceptance test is also one of the weaknesses of Scheme III, in the sense that it must be complete and correct in order for the architecture to satisfy the requirements.

Appendix C

Survey Summary

As part of this effort a small scale survey was conducted of current software development practices in environments serving safety critical applications. The target organizations included four vendors of nuclear reactor protection systems (designated as N1 through N4) and four general software development or audit organizations (designated G1 through G4). Much of the information obtained was identified as proprietary. The following tables summarize non-proprietary information pertinent to the scope of this report. Table C.1 represents characteristics of recently developed safety grade software; this information was solicited only from the nuclear vendors. Table C.2 represents technology that is applicable to the general as well as the nuclear applications and includes information from all eight contacts.

Table C.1: Survey Summary --- Class 1E Products

Characteristic	N1	N2	N3	N4
Source language	PL/M	C(?)	C(?)	PL/M(1)
Future language	C	C	C	C
Size of code	85K	100K		> 64K
Chip type	80286	obsol.	4004	8086(2)
Criteria for test termination		branches	statistics	functions + statistics
First baseline	Requirements	Tested modules	Complete system	Proper methodology
V & V activities	3 steps	2 steps	2 steps	
Personnel Qualification --- Development	Informal	Informal	Informal	Informal
Personnel Qualification --- Test	Informal	Formal training	Formal training	Training
Hardware fault tolerance	4 channel	4 channel	redund.	4 channel
Software fault tolerance	none	none	none	none
Software fault detection	assertion	self-check	self-check	assertion

(1) also Assembly86 and some Pascal.
(2) also Zylog and Motorola chips for auxil. proc.

Table C.2: Survey Summary --- Technology

Characteristic	N1	N2	N3	N4	G1	G2	G3	G4
Cost & schedule models	1	0.5	1.5	1	2^-	2	2^-	1^+
Technical management	0.5	+	+	0.5	1.5	2	2	1^-
Requirements analysis	+	0	1.5	+	2^-	1^-	0.5	*
Configuration management	NQA-2	NQA-2	1^+	1.5	2^-	*	1^-	1^-
Software design & implementation	1	+	1^+	1.5	*	1.5	1.5	1^-
Software integration test	+	0	1	+	*	1	1.5	1^-
Formal methods	0	0	+	0	2	0	0	1^-
Safety & reliability analyses	1^+	1^+	1	0.5	1^+	1^+	1.5	*
Hardware fault tolerance	1^+	1^+	1	1	2	2	*	1^-
Software fault tolerance	0	0	0	0.5	2	1.5	*	1^-

Legend:

Nx = Nuclear vendor
Gx = General software organization
0 = no knowledge of this methodology
1 = has knowledge but has not applied it
2 = has applied it successfully
+ = slight increment
0.5 = significant increment
* = not applicable to this environment or not covered in survey visit
NQA-2 = A standard provides comprehensive guidance for configuration management but little specific guidance for software; its use is not easily represented by the criteria listed above.

Bibliography

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Prentice Hall, 1988.

[2] G. Booch, *Object Oriented Design with Applications.* Redwood City, CA: Benjamin/Cummings, 1991.

[3] Ada 9X Mapping/Revision Team, *Ada 9X Mapping: Volume I - Mapping Rationale, Volume II - Mapping Specification.* Intermetrics, Inc., Cambridge, MA, Version 3.1 ed., Aug. 1991.

[4] C. Anderson, "Ada 9x project report to the public," technical report, Eglin AFB, Florida, Aug. 1991.

[5] Intel Corporation, *PL/M Programmer's Guide*, 1987.

[6] H. Curnow *et al.*, "The synthetic whetstone benchmark," *Computer Journal*, pp. 46-49, 1976.

[7] "Product reviews," *IEEE Computer*, vol. 23, no. 11, pp. 94-97, 1990.

[8] A. Hook *et al.*, "Cost benefit study on ada, institute for defense analysis," Technical Report IDA-P-1930, June 1986. available from: NTIS (703)487-4650, AD-A175 748.

[9] D. Dikel *et al.*, "Commercial market potentials for the use of ada language, institute for defense analyses" Technical Report IDA-M-106, Aug. 1985. available from: NTIS (703)487-4650, AD-A163 273.

[10] L. K. Mosemann, "Ada and C++: A business case analysis," technical report, US Air Force, Washington, DC, July 1991.

[11] W. W. Royce, "Managing the development of large software systems: Concepts and techniques," in *Wescon*, August 1970.

[12] B. W. Boehm, "A spiral model of software development and enhancement," *IEEE Computer*, vol. 21, pp. 61-72, May 1988.

[13] D. R. Wallace and J. C. Cherniavsky, "Guide to software acceptance," NIST Special Publication 500-180, National Institute of Standards and Technology, Washington, DC, 1990.

[14] W. E. Howden, "The theory and practice of functional testing," *IEEE Software*, vol. 2, no. 5, pp. 6-17, 1985.

[15] J. C. Huang, "An approach to program testing," *ACM Computing Surveys*, vol. 7, pp. 113-128, Sept. 1975.

[16] M. Barnes *et al.*, "Software testing and evaluation methods: final report the STEM project," HPR-334, Institutt for Energiteknikk, Halden, Norway, 1988.

[17] P. G. Bishop, *Techniques Directory*, vol. 3 of *Dependability of critical computer systems.* Elsever Applied Science, 1988.

[18] J. W. Duran and S. C. Ntafos, "A report on random testing," in *Proc. 5th International Conference on Software Engineering*, pp. 179-183, 1981.

[19] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Software Engineering*, vol. SE-11, pp. 367-375, Apr. 1985.

[20] D. J. Richardson and L. A. Clarke, "Partition analysis: A method combining testing and verification," *IEEE Trans. Software Engineering*, vol. SE-11, no. 12, pp. 1477-1490, 1985.

[21] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Software Engineering*, vol. SE-6, no. 3, pp. 247-257, 1980.

[22] W. E. Howden, "Weak mutation testing and completeness of program test sets," *IEEE Trans. Software Engineering*, vol. SE-8, no. 4, pp. 371-379, 1982.

[23] L. A. Clarke ar d D. J. Richardson, *Symbolic evaluation methods --- Implementations and applications*, pp. 65-102. Amsterdam, Holland: North-Holland, 1981.

[24] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, 1987.

[25] W. H. Farr and O. D. Smith, "Statistical modeling and estimation of reliability functions for software (SMERFS) user's guide," NSWC TR 84-373, Naval Surface Weapon Center, Dahlgren, VI, 1985.

[26] "Application criteria for programmable digital computer systems in safety systems of nuclear power generating stations," ANSI/IEEE-ANS-7.4.3.2-1982, American Nuclear Society and Institute of Electrical and Electronics Engineers, Inc., 1982.

[27] P. K. Joannou, J. Harauz, *et al.*, "Standard for software engineering of safety critical software," 982C-H 69002-0001, Ontario Hydro and AECL CANDU, Ontario, Canada, 1991.

[28] A. Avižienis, "The four-universe information system model for the study of fault-tolerance," in *Digest of 12th International Symposium on Fault-Tolerant Computing*, pp. 6-13, June 1982.

[29] "Analysis and evaluation of operational data," 1990 Annual Report, U. S. Nuclear Regulatory Commission, July 1991.

[30] D. E. Eckhardt and L. D. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1511-1517, Dec. 1985.

[31] J. C. Knight and N. G. Leverson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Trans. Software Engineering*, vol. SE-12, pp. 96-109, Jan. 1986.

[32] S. S. Brilliant, J. C. Knight, and N. G. Leverson, "Analysis of faults in an n-version software experiment," *IEEE Trans. Software Engineering*, vol. SE-16, pp. 238-247, Feb. 1990.

[33] H. Hecht, "Practicable software fault tolerance for u. s. army tactical systems," Final Report under Contract DAAB07-90-C-B807, SoHaR Incorporated, Beverly Hills, CA, June 1991.

[34] A. Avižienis, "A design paradigm for fault-tolerant systems," in *Proceedings of AIAA Computers in Aerospace VI Conference*, (Wakefield, MA), Oct. 1987.

[35] J. M. Wing, "A specifier's introduction to formal methods," *IEEE Computer*, vol. 23, pp. 8-24, September 1990.

[36] W. P. deRoever, "Foundations of computer science: Leaving the ivory tower," *Bulletin of the European Association for Theoretical Computer Science*, June 1991.

[37] C. Landwehr, J. McLean, and C. Heitmeyer, "Defining formalism (letter to the editor)," *Communication of the ACM*, October 1991.

[38] J. V. Hill, P. Robinson, and P. A. Stokes, *Safety Critical Software in Control Systems*, pp. 92-96. Lecture Notes in Computer Science, Inst. of Electrical Engineering, Stevenage, UK, 1990.

[39] D. L. Parnas, G. J. K. Asmis, and J. D. Kendall, "Reviewable development of safety critical software," in *Proc. International Conference Control and Instrumentation in Nuclear Installations*, (Glasgow, UK), May 1990.

[40] D. L. Parnas, G. J. K. Asmis, and J. Madey, "Assessment of safety critical software," in *Proc. IEEE Denver Section 9th Annual Software Reliability Symposium*, (Colorado Spring, CO), May 1991.

[41] M. Srivas and M. Bickford, "Formal verification of a pipelined microprocessor," *IEEE Software*, vol. 7, pp. 52-64, September 1990.

[42] R. L. London, "Proof of algorithms, a new kind of certification," *Communication of the ACM*, June 1970.

[43] M. Foley and C. A. R. Hoare, "Proof of a recursive program: Quicksort," *Computer Journal*, pp. 391-395, November 1971.

[44] B. Elspass *et al.*, "An assessment of techniques for proving program correctness," *ACM Computing Surveys*, June 1972.

[45] B. W. Boehm, *Software Engineering Economics*. Prentice Hall, 1981.

[46] K. C. Kang, "Requirements engineering project." Briefing charts, April 1990.

[47] P. Zave, "An insider's evaluation of paisley," *IEEE Transactions on Software Engineering*, vol. SE-17, pp. 212-225, March 1991.

[48] S. L. Gerhart, "Applications of formal methods: Developing virtuoso software," *IEEE Software*, vol. 7, pp. 7-10, September 1990.

[49] J. V. Hall, *Software Development Methods in Practice*. Elsevier, 1991.

[50] B. Hanrahan *et al.*, "Requirements analysis and design tool report," technical report, Hill Air Force Base, 1991.

[51] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Engineering*, vol. SE-2, pp. 308-320, Dec. 1976.

[52] M. H. Halstead, *Elements of Software Science*. New York: Elsevier North-Holland, 1977.

[53] K. S. Tso, "Complexity metrics for avionics software," Final Report under WL/AAAF-3 Contract F33615-91-C-1753, SoHaR Incorporated, Beverly Hills, CA, Oct. 1991.

[54] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Software Engineering*, vol. 17, pp. 900-910, Sept. 1991.

[55] U. Mondal *et al.*, "Application of a microprocessor based trip meter in nuclear reactor shutdown systems," in *EPRI Seminar Presentation*, 1992.

[56] M. Hecht, J. Agron, H. Hecht, and K. H. Kim, "A distributed fault tolerant architecture for nuclear reactor and other critical process control applications," in *Digest of 21st International Symposium on Fault-Tolerant Computing*, (Montreal, Canada), pp. 3-9, June 1991.

[57] K. H. Kim and H. O. Welch, "Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications," *IEEE Trans. Computers*, vol. 38, pp. 626-636, May 1989.

[58] S. W. Haeberlin *et al.*, "A handbook for value-impact assessment," NUREG/CR-3568, U. S. Nuclear Regulatory Commission, Dec. 1983.

*U.S. GOVERNMENT PRINTING OFFICE

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.